Programming Language Specifications and Environments

Yves Bertot

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

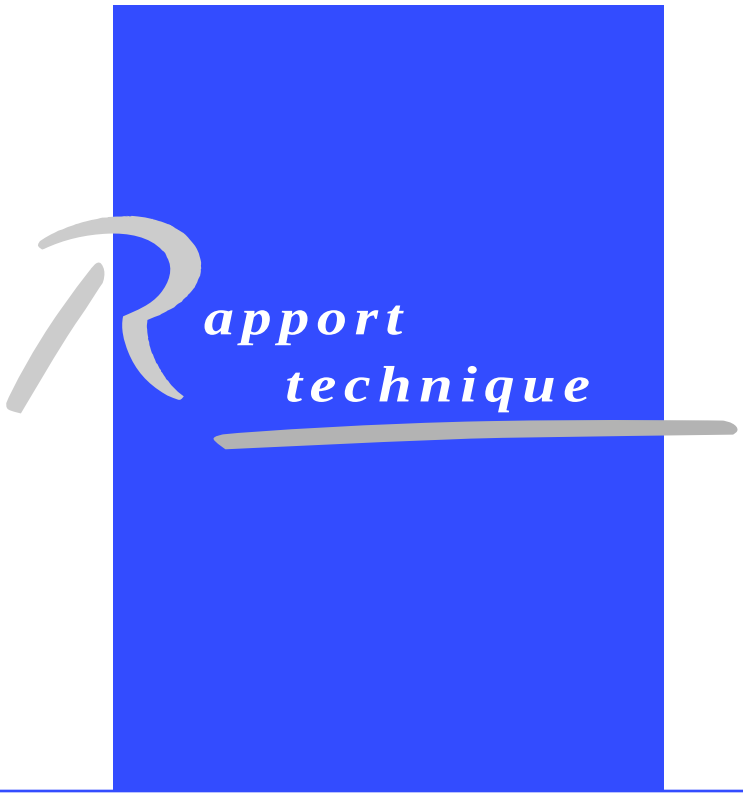
Programming Language Specifications and Environments

Y. Bertot

N° 0212

December 1997

THÈME 2



ISSN 0249-0803

Programming Language Specifications and Environments

Y. Bertot

Thème 2 — Génie logiciel
et calcul symbolique
Projet CROAP

Rapport technique n° 0212 — December 1997 — 47 pages

Abstract: These are course notes on programming language formal descriptions and their use to build programming environments and to prove properties of these languages.

Key-words: Programming languages, Semantic specification, Programming Environment, Centaur, Proof, Coq

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 93 65 77 77 - International : +33 4 93 65 77 77 — Fax : 04 93 65 77 65 - International : +33 4 93 65 77 65
à partir du 01/01/1998
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 — Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

Spécifications de langages de programmation et environments

Résumé : Ce document réunit des notes de course sur la description formelle des langages de programmation et l'utilisation de telles descriptions pour construire des environnements de programmation et prouver des propriétés des langages étudiés.

Mots-clés : Langages de programmation, Spécification sémantique, Environnements de programmation, Centaur, Preuve, Coq

The reasons for working on the formal specification of a programming language may be numerous. The goal may be to provide a definition of the language that is not attached to the implementation on a particular machine. Another aim may be to provide a prototype implementation, where the coding is declarative enough to be acceptably machine independent. Yet another possibility may be to provide a sound basis for users to state and prove judgements about programs.

Two important areas of computer science have shown some interest in formal specifications. One area is that of programming environment generators, where one is interested in deriving programming tools from an abstract description of the programming language. This approach makes it possible to share tools between languages, like error reporting for instance. Users of the programming environment generators can thus provide short descriptions of the tools they want to provide and the actual implementation of these tools is obtained by some compilation process from these descriptions.

In the other area interested in formal specifications, the central tool is a proof tool rather than a programming environment. Here, the actual derivation of programming tools is less relevant than the ease with which one will describe a programming language, feed this description to a theorem proving system, and get this theorem proving system to produce a proof for statements about the language.

Because of the distance between these two areas, there is no insurance that the same notion of "formal specification" is shared by speakers from both sides. After all, if formal specifications are only used to generate programming tools through a compiler, there is no reason these specifications should be easy to use for proving statements. Likewise, the languages used in proofs can be arbitrarily remote from practical languages and there may be no means to extract anything executable from the description used to reason about a language. The work described in these notes attempts to reunite the two sides: we are going to provide formal specifications for a programming language that will be both executable and abstract enough to reason formally. Also, our formal reasonning will be machine-checked. In this respect, this document is also an experience notebook. Most of the specifications and the proofs mentionned in this document are given in Annex.

This work is largely inspired by the book of G. Winskel [14], although we have put a specific emphasis on the generation of tools derived from formal specifications. In some respect, this work also advertises the use of the Centaur and Coq systems for the study of programming languages, providing an environment where activities ranging from actual program editing and manipulation to correctness proofs can be envisionned in an integrated manner [2]. Following this point of view, we have deliberately limited our study to Natural Semantics, setting aside other possible forms of formal specification. Thus, readers more interested in *denotational* semantics or *axiomatic* semantics are invited to consult the book [14].

1 Starting little

In this section, we present the formal specification of a small programming language, little, that handles boolean and integer values with only basic operations (for example, multiplication is not provided as a basic operator). The following program is actually a program that computes the number factorial of 3 (3!).

```
declarations
variable x : integer = 3;
variable i : integer = 0;
variable intermediary : integer = 0;
variable result : integer = 1
in
while x > 0 do
  begin
  i := x;
  intermediary := result;
  result := 0;
```

```
while i > 0 do
  begin
  result := result + intermediary;
  i := i - 1;
  end;
x := x - 1;
end
```

We first describe the specifications of the language related to its syntactical aspects.

1.1 Abstract Syntax

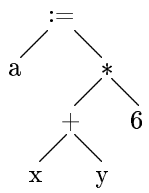
Traditionnally, programming languages are defined by their syntax, that is, the set of text documents that represent correct programs. But the point of view of identifying a program by its text gives too much emphasis to details such as indentation and extraneous parentheses. For example, the two program fragments

```
a:=(((x+y)))*6;
```

and

```
a:=
   (x+y)*6
;
```

are basically the same program and should not be distinguished. The only information that matters on these fragments is that they are the assignment of the value of an expression to a variable, where the expression is the product of the sum of two variables by an integer. This information can sensibly be described by a tree-like representation like the following one:



In a textual manner, this representation may also be written the following way:

assign(ident a, times(plus(ident x, ident y),int 6))

Having decided to represent programs as trees, we still have to represent syntactic constraints: any tree will not necessary represent a relevant program. For example, assignment instructions are only made of two parts and the left part must necessarily be an identifier. The specification of all the constraints that trees must verify to represent program fragments is called an *abstract syntax*. In the rest of this section, we describe a formalism for describing abstract syntaxes.

An Abstract Syntax Formalism An abstract syntax is made of *phyla* and *operators*. Phyla represent tree categories, for instance instructions or expressions, while operators represent primitive tree patterns.

There are three classes of operators: fixed arity operators, list operators, and atomic operators. *Fixed arity operators* represent conventional tree patterns. For instance, the assignment instruction of the language little is represented by the assign operator, described in the following manner:

assign → ID EXP

This description expresses that this operator has two sub-trees and that trees accepted as first child are not in the same category as trees accepted as second child.

List operators describe program patterns where the same syntactic category is repeated several times. In fact, there exists two types of list operators, depending on whether empty lists are accepted or not. In the language little declaration lists are described by the decls operator:

decls → DECLS +...

This description expresses that at least one declaration is necessary. The language also contains an operator to represent instruction sequences, described in the following manner.

sequence → INST *...

This description expresses that empty sequences are accepted.

The *atomic* operators describe elementary constructions of programs: boolean values (true and false), identifiers (used as variable names), ,immediate integer values, and types (integer) and (boolean).

All operators are sorted in phyla. Each phylum is defined as the set of head operators accepted for trees in this phylum. It is considered normal that two distinct phyla contain a common operator. It is even possible to express explicitely that one phylum is included in another. For instance, the phylum EXP is described the following way:

EXP ::= VAL ID and or plus minus gt eq

This description expresses that the phylum for immediate values, VAL, and the phylum for identifiers, ID, are included in the phylum for expressions.

On phylum Inclusion The question whether different phyla should be allowed to contain common operators is a much debated subject. Many systems for designing programming language tools, among them the Program Synthesizer Generator [12] developped at Cornell University, or the Attribute Grammar Evaluator FNC2 developped at INRIA, insist that common operators should simply be rejected. The main advantage of this restriction is that it transforms abstract syntax specifications into simple data type declarations like those found in the ML programming language and that computing the type of an expression in that setting is a well understood process. However, this restriction reduces the level of abstraction available in the representation of program fragments. When no phylum inclusion is permitted, language designers are forced to introduce new "dummy" operators that represent explicitly the coercion of terms from one phylum to another. For example, the phylum EXP given above is described by the following line in this new setting:

EXP ::= val_to_exp id_to_exp and or plus minus gt eq

where the operators val_to_exp and id_to_exp have the following description:

val_to_exp → VAL
id_to_exp → ID

With this abstract syntax specification, the representation of the program fragment a:=(x+y)*6 becomes the following one:

assign(ident a, times(plus(id_to_exp(ident x), id_to_exp(ident y)), val_to_exp(int 6)))

We argue that the new abstract syntax is less "abstract" than the previous one. In the following we shall stick to a style that takes advantage of phylum inclusion. This decision may have consequences when translating the language specifications into input for other tools such as an attribute grammar evaluator or a theorem prover, where abstract syntaxes may be more restricted.

1.2 A Practical Use to a Syntax Specification

By putting the accent on the tree structure of programs, we also emphasize the need for tools to manipulate such tree representations, also called *structure editors*. It is interesting to compare the use of this kind of editors with regular text editors to understand what is gained and what is lost when changing from one tool to the other. In a text editor, the user can move a cursor around, select regions by setting a begin mark and an end mark, copy or cut regions, insert the last copied region before or after the current position of the cursor. Inserting new characters can be done quite freely, and documents can be incoherent with respect to any syntax rules.

In a structure editor, the notion of cursor does not exist: one always select a region, and this region always correspond to a complete syntactic construct. Copying a program fragment can be done only for complete syntactic constructs and inserting and cutting are available only in very restricted conditions (when

the father is a list operator). Structure editors also provide the possibility to replace expression by patterns of some language constructs, with holes, also called place-holders, that will have to be filled in. This notion of "a hole to fill in" does not exist in text editors, but it somehow replace the possibility to manipulate incomplete programs.

Text editors like Emacs [3] provide facilities for defining little automata to perform repetitive tasks or regular expression replacements. Structure editors can also be fitted with macro languages oriented toward the manipulation of tree structures, like the language mentol provided in Mentor or transformation menus provided in the Synthesizer Generator and Centaur systems.

For instance, we can consider the transformation that replaces an instruction of the form:

$$\text{if } a \text{ then } \mathit{fragment}_1 \text{ else } \mathit{fragment}_2$$

by an instruction of the following form, only when a is an identifier:

$$\text{if not}(a) \text{ then } \mathit{fragment}_2 \text{ else } \mathit{fragment}_1$$

This transformation is specified by a rule of the following form:

```
'if to if not' : if(*a^ID, *inst1, *inst2) : {INST}
  --> if(not(*a),*inst2,*inst1)
```

The first part of this rule is a symbolic name, The second part, if(*a^ID,*inst1,*inst2), is a tree pattern that indicates those trees that can be transformed. In this pattern, *a^ID, *inst1, and *inst2 are variables that match subterms of the tree to transform. Note that the notation *a^ID enables the user to express that only those if statements where the conditionnal is an identifier will be transformed. The third part {INST} indicates the context in which the transformation can be performed. Finally, the fourth part, if(not(*a),*inst2,*inst1), describes the tree produced by the transformation. This tree contains occurrences of the variables *a, *inst1, and *inst2 for the corresponding subterms of the initial tree to be copied in the corresponding places. It is not necessary that the variables occurring in the right hand side of the rule also occur in the left hand side. When a new variable occurs, this only indicates that a new empty place holder will be created. A programming environment might provide a menu of transformations where this rule would appear. Such a menu makes it possible to trigger a transformation by simply clicking with the mouse.

1.3 Extension: Higher-order Abstract Syntax

Generally, the name of variables used in programs is important only for readability. The only thing that matters being that the same variable is used at different locations in the program. For example, the following two program fragments are so similar they could be replaced by one another in any context without changing the behavior of the embedding program.
fragment 1:

```
x:= 10;
y:=0;
while (x < 0) do
  begin
    result := result + x;
    x := x - 1;
  end;
```

fragment 2:

```
y:= 10;
x:=0;
while (y < 0) do
  begin
```

```
    result := result + y;
    y := y - 1;
  end;
```

One may wonder if there exists a way to formally describe the syntax of our programming language that enables us to view the two fragments as syntactically equivalent. The answer is that there exists such a formalism, based on the view of programs as functions over the names of variables occurring in these programs. Thus, two programs with different variable names may be viewed as identical programs as long as they can be obtained by applying one such function to different sets of variables. Because this formalism makes a heavy use of functions, it is often called *higher-order abstract syntax*[6]. It is unclear how well higher-order abstract syntax can be used in a structure editor. To our knowledge there is no structure editor based on higher-order abstract syntax available yet. However, higher-order abstract syntax may prove useful to simplify the task of reasoning on programs.

1.4 Concrete Syntax

The emphasis put on abstract syntax so far does not preclude a useful connection to concrete syntax and the parsing of textual documents.

1.4.1 Connecting to a Context Free Grammar

The *concrete syntax* of a language describes the set of constraints that textual documents must verify to correctly represent programs. For a programming environment with structure editors, it is also necessary to compute the tree representation of programs. Concrete syntax specifications can be used to generate a tool, a *parser*, that computes the correspondence between textual documents and trees of abstract syntax.

A concrete syntax is given by a *context-free* grammar composed of *productions*. To obtain a parser, one must also annotate each production with the specification of the tree to construct when recognizing this production. For example, the syntax of assignments may be described by a production of the following form:

⟨instruction⟩ ::= ⟨ident⟩ ":=" ⟨exp⟩

Apart from the equation sign "::=", the elements of this rule represent text fragments. This production expresses that an instruction (represented by ⟨instruction⟩) can be recognized when there is an identifier (represented by ⟨ident⟩), followed by :=, and followed by an expression (represented by ⟨exp⟩). In the parser specification, this rule is annotated with a tree construction command

assign(⟨ident⟩, ⟨exp⟩).

This command expresses that when a program fragment is recognized using this rule, one must associate to this fragment a tree with assign as head operator, the tree associated to ⟨ident⟩ as the first child, and the tree associated to ⟨exp⟩ as the second child. Note that the terms ⟨ident⟩ and ⟨exp⟩ represent two different kind of objects: text fragment when they appear in a production, trees when they appear in a tree construction command.

1.4.2 Designing a Multiple Entry Point Grammar

Traditionnally, a context free grammar is given with a single entry point, meaning that one is interested in parsing complete documents (complete programs, in the case of programming languages). This is sufficient when the parser derived from the syntax specification is to be embedded in a compiler, which takes complete programs to produce complete executable files. In the context of programming environments, multiple entry points are desirable, as one may want to fit structure editors with the possibility to also edit sub-expressions as text fragments that have to be reparsed to recover the tree structure.

To provide this functionality, it is necessary to implement a multiple entry parser, which will be enable to analyze not only complete programs but also program fragments corresponding to each phylum of the abstract syntax. A first step is to figure out which non-terminal variable in the grammar recognizes exactly the text fragments corresponding to the terms accepted in each phylum. Very often, this task is simple

because the concrete syntax is close enough to the abstract syntax. In complicated cases though, it may be necessary to add new non-terminal variables and new productions to the grammar specification. When this task is complete, one only needs to add productions of the following form to the grammar specification:

⟨start⟩ ::= T_p⟨N⟩;
⟨N⟩

For each phylum p, where T_p is a new token associated to p that cannot appear in regular programs and ⟨N⟩ is the non-terminal variable associated to the phylum p.

For example, in the case of the language little, one adds the following rule to the syntax specification.

⟨start⟩ :: [INST]⟨instruction⟩;
⟨instruction⟩

1.4.3 Using a "black box" parser

Sometimes, it may be useful to connect a structure editor to a language that already has a parser, without having to rewrite a complete specification of the concrete syntax. This is practicable if the parser actually produces a tree-like structure that can be transformed into an abstract syntax defined in our formalism. In this case, the structure editor will only use the abstract syntax definition of the formalism and no concrete syntax definition.

A common problem in this setting is that the parser may not provide the complete set of entry points necessary for in-place editing. This kind of limitation can be turned around using a *context information table*.

A context information table consists in an unorder lists of tuples, where each tuple contains a phylum name, a *pre-string*, a *post-string*, and a path. The *pre-string* and the *post-string* should be specific to the phylum, in the sense that the following properties should be verified:

1. If ⟨text⟩ is the text associated to any tree in the phylum, then the document obtained by concatenating the pre-string, ⟨text⟩ and the post-string should be accepted by the black box parser.

2. If ⟨text⟩ is a text such that the concatenation of the pre-string, ⟨text⟩, and the post-string is accepted by the black box parser, then ⟨text⟩ should be related to a tree that is a member of the phylum. Moreover, this tree should appear in the result of the parser as the subtree given by the path.

The notion of *path* has been left unspecified. There are several ways to implement such an object. A path can be a list of integers, indicating tree navigation commands from the root to a specific subtree. A path can also be a variable and a linear pattern (that is, a pattern where some subterms are variables, with all variables being paiwise distinct).

For instance, the following tuple could could be used to give the context information for the phylum ID in the language `little` (the path, here represented by (*x,program(decls(decl(*x,*y),*z))) could also be represented by the sequence [1,1,1]).

```
(ID, "declarations variable ", "= 1 in begin end",
     (*x,program(decls(decl(*x,*y),*y))) )
```

1.5 Formatting

Even in structure editors, the tree representation of programs is not comfortable. To make documents more readable, one must provide ways to produce textual documents from abstract syntax trees. The production of textual documents is specified using the PPML [8] language. This formalism describes the following aspects of formatting:

- indenting instructions and expressions,
- line break strategies when pages are not wide enough,
- an ellipsis mechanism that make it possible to choose a level of detail,

- the expression chosen when a user designates a token on the screen,
- choices of colors and character fonts to highlight keywords and important expressions,

This formalism uses rules of the following form:

$$pattern \rightarrow format$$

The lefthand side of the rule is a tree pattern that possibly contains variables and the righthand side contains formatting instructions where variables may appear representing recursive calls of the formatter on the corresponding sub-expressions.

For instance, the rule for formatting an assignment has the following form:

```
assign(*id,*exp)   →   [ <hv 1,1,0> [<h 1> *id ":="] *exp];
```

According to this rule, the formatter will display an assignment by first displaying the first child, then the sign :=, and then the second child. Square brackets, [and], and the separators, <hv 1,1,0> and <h 1> provide indenting and line breaking information: the sign := must always be on the same line as the first child, separated by one unit of space, while the second child can be displayed on the next line, indented with one unit of space, when the text is too long to fit on the same line.

2 Semantic Aspects

To describe the semantics of a programming language, we use a formalism called *Natural Semantics* based on relations and inference rules to reason about these relations. The relations that one describes are properties involving programs or program fragments. An example of such properties may be *in environment ρ, instruction I executes normally and returns a new environment ρ'*. This example is used for describing the *dynamic* or *operational* semantics of a language, by expressing how instructions act on their environment. Other properties may state facts related to type checking, compilation, etc.

The inference rules have the following form, where the *premises* and *conclusion* are relations between abstract syntax trees:

$$\frac{premise_1 \quad \cdots \quad premise_n}{conclusion}$$

The meaning of such a rule is that the conclusion holds if all the premises hold. Most of the time, the relations used as conclusion and premises have a privileged argument, called the *subject*, and each of these relations expresses that some property holds for its subject. Also, the subject of each premise is usually a subterm of the subject of the conclusion. This custom, although it is not always respected, makes that this style has also been dubbed *structural operational semantics* in a seminal paper of G. Plotkin [10].

Semantic specifications can be used to generate tools exactly the same way as abstract syntax specifications were basically used to define structure editors. So far, two ways of executing semantic specifications have been explored. The first method is based on a strong analogy between Natural Semantics rules and Horn clauses. Indeed, every inference rule can be compiled as a Prolog rule and the various abstract syntax trees can be compiled as Prolog terms. A Prolog interpreter can then be used to verify that some relations are provable. Thus, questions of the form *is my program correct with respect to type constraints?* or *can my program be executed?* can be answered to by the computer. For the second question, the answer will even contain the result of the computation. Hooking this kind of computation to our structure editors is rather simple. The editor can be fitted with a menu proposing various tools to run on programs. When the user chooses an option from this menu, the program in the window is translated into a Prolog term and the proof of the relevant Prolog formula is required from a Prolog interpreter. When the proof is completed, the value variables in the Prolog formula that have been instantiated during computation can be retrieved and displayed to the user using a window of the structure editor.

The second method of executing semantic specifications is based on the structural property of semantic specifications[1]. According to this point of view, every relation used in a specification expresses that some attributes are carried by a privileged argument of the relation: the subject. Each inference rule specifies the

relations between the attributes of a node in an abstract syntax tree, some attributes carried by its parent node (these attributes are called *inherited* attributes), and some attributes carried by its children nodes (these attributes are called *synthesized* attributes). Formal specifications based on this kind of attributes are called *attribute grammars*. A key aspect of attribute grammars is that their execution can be incremental[11]. If the user manipulates a program in the structure editor, runs a tool on this program, and modifies slightly the program, it is possible to re-run the tool while recomputing only the attributes carried by the nodes that are affected by the changes provoked by the user. With this technology, it is possible to have the tool perform only a few computations at every modification of the edited program and keep the user aware of the result of these computations. For instance, this is especially useful if the tool is a type-checker that will notify the user as soon as a type incoherence is introduced in the program. The Synthesizer Generator[12] is a famous example of a structure editor based on this idea.

It should be noted however, that none of these compilation techniques actually respect faithfully the meanings of semantic specifications. Using a Prolog interpreter to execute formal specification helps us in finding one solution to problems, but the solution that is found is chosen by the Prolog strategy. This is a weakness when one wants to study the non-determinism of some programming languages, for instance in the case of concurrent languages. As far as compilation to attribute grammars goes, the restriction is even stronger: only some classes of semantic specifications can be compiled to attribute grammar specifications.

All the semantic specifications we describe in this paper are actually used to generate a tool. Thus, all these specifications can be viewed as programs written in a programming language called Typol [9]. The fact that specifications can be executed will have a significant impact on our design. For instance, executing a specification with a specific program as input will enable us to test the coherence of the specification.

2.1 Type Discipline

An example of a property that can be expressed is the property for programs to respect a type discipline. For our programming language little, we will consider a program to be *well typed* if every variable used in the program is declared in the initial list of declarations, if the initial value has the type (boolean or integer) given in the declaration and if every usage of this variable has the same type.

In our specification, we use the following syntactic form to express that a program P is well typed:

$$\vdash P,$$

the following form to express that a list of declaration is coherent with respect to type discipline:

$$\vdash D,$$

the following form to express that an instruction I is well typed with respect to a given environment ρ:

$$\rho \vdash I,$$

and the following form to express that an instruction E is well typed with type τ in a given environment ρ:

$$\rho \vdash E \ : \ \tau$$

Thus, the rule for checking that a complete program is well typed has the following form:

$$\frac{\vdash D \qquad D \vdash I}{\vdash \#program(D, I)}$$

In this rule, the sign # is used to avoid mistaking the operator name *program* with a keyword from the Typol language. The rule expresses that a program is well typed if the instruction it contains is well-typed with respect to the list of declarations.

The rule for checking that an assignment is well typed has the following form:

$$\frac{\rho \vdash Id \ : \ \tau \qquad \rho \vdash E \ : \ \tau}{\rho \vdash \mathsf{assign}(Id, E)}$$

$$\text{exec\_if\_true:}\quad \frac{D \vdash E \mapsto \mathsf{true} \qquad D \vdash I1 \to D'}{D \vdash \mathsf{if}(E, I1, I2) \to D'}$$

$$\text{exec\_if\_false:}\quad \frac{D \vdash E \mapsto \mathsf{false} \qquad D \vdash I2 \to D'}{D \vdash \mathsf{if}(E, I1, I2) \to D'}$$

$$\text{exec\_while\_false:}\quad \frac{D \vdash E \mapsto \mathsf{false}}{D \vdash \mathsf{while}(E, I) \to D}$$

$$\text{exec\_while\_true:}\quad \frac{D \vdash E \mapsto \mathsf{true} \qquad D \vdash I \to D' \qquad D' \vdash \mathsf{while}(E, I) \to D''}{D \vdash \mathsf{while}(E, I) \to D''}$$

$$\text{sequence\_end:}\quad D \vdash \mathsf{sequence}[] \to D$$

$$\text{sequence\_rec:}\quad \frac{D \vdash I1 \to D' \qquad D \vdash I2 \to D''}{D \vdash \mathsf{sequence}[I1.\, I2] \to D''}$$

$$\text{exec\_assign:}\quad \frac{D \vdash E \mapsto V \qquad D \overset{\text{update}}{\vdash} I \backslash V \to D'}{D \vdash \mathsf{assign}(I, E) \to D'}$$

Figure 1: The rules for instruction execution

This rule expresses that an assignment is well typed if the assigned variable is defined in the list of declarations ρ, with a given type τ and if the evaluated expression is well typed with respect to this list of declarations, with the same type τ.

It is possible to provide another version of the type-checker where the output of an execution will be a list of error messages referring to positions in the program. One has to take care to respect the same notion of well-typedness.

2.2 Dynamic Semantics

We can use the same kind of inference rules to specify the *dynamic semantics* of the programming language. We consider that the result of executing a program is the complete list of variables along with their values. Such a list can be represented by a list of declarations as already found in the abstract syntax of the language. We specify the dynamic semantics with a property of the form

$$\vdash P \to D$$

that expresses that the execution of program P terminates and returns a list of declarations D, a property of the form

$$D \vdash I \to D'$$

that expresses that the execution of an instruction I in the environment given by D terminates and returns a list of declarations and valuesD', and a property of the form

$$D \vdash E \mapsto V$$

that expresses that the evaluation of expression E in the environment given by D returns the value V.

For instance, the rule describing the execution of a program has the following form:

$$\frac{D \vdash I \to D'}{\vdash \#\mathsf{program}(D, I) \to D'}$$

This rule expresses that if the execution of I terminates in the environment D and returns the environment D' then the execution of $\mathsf{program}(D, I)$ terminates and returns the same environment.

The rule describing the execution of a sequence has the following form:

$$\frac{D \vdash I_1 \rightarrow D_1 \qquad D_1 \vdash I_2 \rightarrow D_2}{D \vdash \mathsf{sequence}[I_1.I_2] \rightarrow D_2}$$

It expresses that the execution of the two instructions has to terminate for the execution of a sequence of two instructions to terminate, and it gives the relation between the various computed environments. We give the whole set of inference rules in figure 1.

2.3 Constant Propagation

The Typol language can also be used to describe other ways to look at programs than simply checking their type coherence or executing them. In this section, we describe an example of program transformation that performs constant propagation. The purpose of this transformation is to simplify programs to take into account the value of variables when this value can be computed statically, that is, without running the program. For example, our tool can perform relevant simplifications on a program fragment that has the following form:

```
a := true;
if a then
 begin  x := 2; y := z + (x * 4); z := x + y; end
else
 x := 0
```

and replace it with the following equivalent fragment, inferring the way the conditional instruction and the various assignments will be performed.

```
a:= true;
x:= 2;
y:= z + 8;
z:= z + 10;
```

As before, we specify constant propagation by describing a number of properties on programs, instructions, expressions, etc. We have a property

$$prop\_eval(s \vdash e \mapsto e')$$

that expresses that the expression e' is obtained from the expression e by simplifying it with respect to a mapping of values to variables given by s.

We have a property

$$s \vdash I \rightarrow I' + s'$$

that expresses that the instruction I' is obtained from the instruction I by simplifying it with respect to the mapping given by s and that s' is a new mapping derived from s and I.

We have a property

$$\text{partial\_update}(s \vdash id \backslash val \rightarrow s')$$

that expresses that s' is the same mapping as s except that it maps the identifier id to the value val.

We also have a property

$$\text{remove}(s \vdash id \rightarrow s')$$

that expresses that s' is the same mapping as s except that it does not associate any value the identifier id. While all the other properties so far had an equivalent in the dynamics semantics of the language, there is

no equivalent for this one: during regular execution, it does not happen that the value of a variable becomes undetermined. But this will happen for constant propagation.

For instance, the rules describing constant propagation over assignments have the following form:

$$\text{prop\_assign\_val:} \quad \frac{\text{prop\_eval}(s \vdash E \mapsto val) \quad \text{partial\_update}(s \vdash id \backslash val \rightarrow s')}{s \vdash \mathsf{assign}(id, E) \rightarrow \mathsf{sequence}[] + s'}$$

$$\text{prop\_assign\_exp:} \quad \frac{\text{prop\_eval}(s \vdash E \mapsto E') \quad \text{remove}(s \vdash id \rightarrow s')}{s \vdash \mathsf{assign}(id, E) \rightarrow \mathsf{assign}(id, E') + s'} \quad \text{provided } E' \text{ is not a value}$$

The Typol language enables users to express that some variable names are reserved for some syntactic categories. It is the case in these rules where the variable val is not just any expression: it can only represent a value, that is, an immediate boolean or integer. The rule prop_assign_val expresses that if the propagation of the constant mapping s reduces the expression E to a value val and if updating the mapping s with this value and this identifier yields the mapping s', then the propagation of constants over the assignment $\mathsf{assign}(id, E)$ reduces to an empty instruction together with the new mapping s'. Intuitively, this rule describes the case where the computation of the assignment can be performed at processing time, the assignment disappears, and the new value computed for the variable is used for processing the instructions that follow.

The rule prop_assign_exp expresses that if the propagation of the constant mapping given by s reduces the expression E to another expression E', which is not a value, then the assignment $\mathsf{assign}(id, E)$ must be transformed into the assignment $\mathsf{assign}(id, E')$ together with a new mapping s' where no value is bound to the identifier id. Intuitively, this rule describes the case where the computation of the assignment cannot be completely performed at processing time, the assignment must remain, and the assigned variable must be undetermined for later instructions.

We can also have a look at the rules for performing constant propagation on expressions. In these rules, we use an abbreviation mechanism that enables us to omit the predicate name prop_eval for each use of this predicate. Thus, we write $s \vdash e \mapsto v$ instead of $\text{prop\_eval}(s \vdash e \mapsto v)$. For instance, the rules for the operator plus are as follows:

$$\text{prop\_eval\_plus\_val:} \quad \frac{s \vdash E_1 \mapsto val_1 \quad s \vdash E_2 \mapsto val_2 \quad \text{add}(val_1, val_2, val)}{s \vdash \mathsf{plus}(E_1, E_2) \mapsto val}$$

$$\text{prop\_eval\_plus1:} \quad \frac{s \vdash E_1 \mapsto V1 \qquad s \vdash E_2 \mapsto V2}{s \vdash \mathsf{plus}(E_1, E_2) \mapsto \mathsf{plus}(V_1, V_2)} \quad \text{provided } V1 \text{ is not a value}$$

$$\text{prop\_eval\_plus2:} \quad \frac{s \vdash E_1 \mapsto V1 \qquad s \vdash E_2 \mapsto V2}{s \vdash \mathsf{plus}(E_1, E_2) \mapsto \mathsf{plus}(V_1, V_2)} \quad \text{provided } V2 \text{ is not a value}$$

The first rule expresses that if the two subexpressions of a sum reduce to values then the value of that sum can be computed at processing time. The other two rules simply express that something remains to be computed if one of the two subexpressions cannot be reduced at compile time.

3 Proving Properties of the Language

Formal specification of programming languages are objects that one should be able to reason about. we show in this section that natural semantics specification lend themselves quite well to reasonning.

We have seen that formal specification of a programming language could encompass several executable tools working on this language. For instance a dynamic semantics describes how to execute programs in the studied language, in other words it is an interpreter.

In some sense a dynamic semantics can be used to express formally the value returned when executing one specific program. However, this should not be the preferred way to prove properties of programs, because other forms of semantics, like axiomatic semantics, are more adapted to this talk.

On the other hand, Natural Semantics is well adapted to reasonning about a language as a whole, thus handling statements where the exact value of programs is left unknown. The e statements one will address in this case will be quantified over programs or program fragments. The following sentences are example of properties that can be treated in this context:

1. Every program terminates (that is, its execution has a finished proof).
2. Two executions of a same program return the same value.

The dynamic semantics of a programming language can also be a yardstick against which other specifications will be matched, yielding statements of the following form:

3. If an expression e is typable with type τ and if this expression evaluates to a value v, then v has type τ (*subject reduction theorem*).
4. If an expression e evaluates to a value v, if e is well typed, if e can be compiled in a program p', if p' can be executed to a value v' (using the target languages' dynamic semantics), then v' is the compilation of v.

In this sense the proofs that one can envision using Natural semantics are proofs of tools working on a language: the statement given in example 2 expresses that the specification is precise enough (when the language is intended to be deterministic), a statement like the one given in example 4 expresses that a compiler is correct.

3.1 Reasonning by induction

Most of the objects handled in our reasoning are of structural nature. Of course, programs have a structural nature, where each node makes it possible to compose terms of smaller size together. We shall also reason over proofs and the existence of proofs. We shall see that these proofs also have a structural nature, with the inference rules of a specification being used to compose proofs of smaller size together. For these two ways of composing objects together, we will use two means of induction reasoning: structural induction and induction on the size of proofs (also called rule induction in [14]).

3.1.1 structural induction

Following Winskel [14], we come back to the principle of induction on natural numbers to illustrate the various principles of induction that we are going to use.

Let $P(n)$ be a property of natural numbers. The principle of induction on natural numbers says that in order to show that $P(n)$ holds for all natural numbers it is sufficient to show the following two properties:

- $P(0)$ holds,
- if $P(n)$ holds then so does $P(n+1)$ for any natural number n.

Using mathematical notations, this principle can be written more succinctly:

```
nat_ind:
  ∀P:nat ⇒ Prop.(P 0)⇒ (∀n:nat. (P n) ⇒ (P (S n))) ⇒ ∀n:nat.(P n)
```

This principle is intuitively clear: If we know $P(0)$ (this is the *base case* and we have a method of showing $P(m+1)$ from the assumption $P(m)$ (this is the *inductive case* and the extra assumption is called the *induction hypothesis*) then from $P(O)$ we know $P(1)$, and applying the method again $P(2)$, and so on. This principle follows the idea that natural numbers are built-up from 0 by repeatedly taking successors.

Inductive sets share this property with the set of natural numbers: trees are built-up from atomic trees by repeatedly applying constructors to previously built trees. Thus, every inductive set can be fitted with a similar induction principle, usually called *principle of structural induction.* Atomic constructors of the language are the equivalent of 0. A proof by structural induction will contain as many base cases as there are atomic operators. Other operators act like the "add one" operator. A proof by structural induction will contain as many induction cases as there are such operators. In each induction case, there will be one induction hypothesis for each subterm: two for a constructor of arity two, and so on.

Many sorted induction

this section is difficult and can be left aside for a first reading.

The situation is made slightly more complicated by the presence of phyla. In this case a proof by induction may need several induction propositions, one per phylum. One easy way to understand proofs by induction over languages with many phyla is to view such a language as a one-phylum language together with a collection of properties $\phi_1, \ldots, \phi_n$, which characterize the phyla. For instance, if *op* is an operator of language, accepting two children taken in the phyla $\texttt{PHYLUM}_i$ and $\texttt{PHYLUM}_j$ and belonging to the phylum $\texttt{PHYLUM}_k$, the property ϕ_k would be defined by a natural semantics specification containing the following rule:

$$\frac{\phi_i(t_1) \qquad \phi_j(t_2)}{\phi_k(op(t_1, t_k))}$$

Of course, the specification would also contain similar rules for all the operators present in the phylum $\texttt{PHYLUM}_k$.

It is then possible to prove a property by induction on all the operators with different propositions for the different phyla, by taking a property P with the following form :

$$(\mathsf{P}\ \mathsf{e}) = ((\phi_1\ \mathsf{e}) \Rightarrow (\mathsf{P}_1\ \mathsf{e})) \wedge \ldots \wedge ((\phi_k\ \mathsf{e}) \Rightarrow (\mathsf{P}_k\ \mathsf{e}))$$

3.1.2 Induction on proof trees

Like the operators of an inductive set, inference rules are the constructors of a set of derivations. This analogy makes it possible to consider a principle of induction that is a structural induction not on abstract syntax trees, but on derivations. We are now going to formalize the principle of induction associated to an inductive definition.

Without loss of generality, we can restrict our study to a predicate of one argument R defined by a finite set of rules $r_1, \ldots, r_n$ of the following form, where P_j is a predicate repressenting side conditions:

$$r_j: \qquad \frac{R(t_j^1) \quad \ldots \quad R(t_j^{l_j}) \qquad P_j(t_j^1, \ldots, t_j^{l_j}, c_j)}{R(c_j)}$$

In these rules, the expressions t_j^i and c_j are schemas representing classes of abstract syntax trees. Now, let us suppose that we have a tree t such that $R(t)$ is provable. If that is the case, there is a finite derivation whose conclusion is $R(t)$ This derivation necessarily finishes with an instance of some rule r_j. Thus, there exist instances $t_j'^1, \ldots, t_j'^{l_j}$, of the schemas $t_j^1, \ldots, t_j^{l_j}$ such that the formulas $R(t_j'^1), \ldots, R(t_j'^{l_j})$, and $P_j(t_j'^1, \ldots, t_j'^{l_j}, t)$ are provable. Obviously there exists derivations for the formulas $R(t_j'^i)$ that are subderivations of the one for $R(t)$. The induction principle will use that property of subderivation with a statement of the following form:

For every property Q on abstract syntax trees, if, for every i in $\{1, \ldots, n\}$, one can prove $Q(c_j)$ from the

facts $R(t_j^1)$, ..., $R(t_j^{l_j})$, $P(t_j^1, \ldots, t_j^{l_j}, c_j)$, *and* $Q(t_j^1)$, ..., $Q(t_j^{l_j})$ *then for every* t *one has* $R(t) \Rightarrow Q(t)$.
In this statement, the formulas $Q(t_j^1)$, ..., $Q(t_j^{l_j})$ are the induction hypotheses for each possible case of inference rule used to complete the derivation. This kind of induction principle is also called rule induction in [14].

3.2 Examples of proofs

3.2.1 update then evaluation in the environment

We suppose that the `update` and `lookup` relations are given by the following four rules:

update_found: $\mathsf{decls}[\mathsf{decl}(Id, t, \_).D] \overset{\text{update}}{\vdash} I\backslash V \rightarrow \mathsf{decls}[\mathsf{decl}(Id, t, V).D]$

update_rec: $$\frac{D \overset{\text{update}}{\vdash} I\backslash V \rightarrow D'}{\mathsf{decls}[\mathsf{decl}(Id', t, V_)'.D] \overset{\text{update}}{\vdash} I\backslash V \rightarrow \mathsf{decls}[\mathsf{decl}(Id', t, V').D']} \quad \text{provided } Id \neq Id'$$

lookup_found: $\mathsf{decls}[\mathsf{decl}(Id, t, V).D] \overset{\text{lookup}}{\vdash} I \mapsto V$

lookup_rec: $$\frac{D \overset{\text{lookup}}{\vdash} I \mapsto V}{\mathsf{decls}[\mathsf{decl}(Id', t, V_)'.D] \overset{\text{lookup}}{\vdash} I \mapsto V} \quad \text{provided } Id \neq Id'$$

We want to prove that an `update` followed by a `lookup` for the same identifier will return the same value. Expressed more formally, this can be stated as follows:

$$\forall D, Id, V, D'. \quad (D \overset{\text{update}}{\vdash} Id\backslash V \rightarrow D') \quad \& \quad (D' \overset{\text{lookup}}{\vdash} Id \mapsto V') \Rightarrow V = V'$$

We will prove this by structural induction on D and D'.
base case: if $D = \mathsf{decls}[]$, then the update and lookup statement are both impossible to realize.
recursive case: Let us suppose that there exist $d_1, Id_1, t_1, v_1, Id_2, t_2, v_2, d_2$ such that
$D = \mathsf{decls}[\mathsf{decl}(Id_1, t_1, v_1).d_1]$ and $D' = \mathsf{decls}[\mathsf{decl}(Id_2, t_2, v_2).d_2]$.

As Induction hypothesis, let us also suppose that d_1 and d_2 are such that

$$(d_1 \overset{\text{update}}{\vdash} Id\backslash V \rightarrow d_2) \quad \& \quad (d_2 \overset{\text{lookup}}{\vdash} Id \mapsto V') \Rightarrow V = V'$$

Now, the statement $(D \overset{\text{update}}{\vdash} Id\backslash V \rightarrow D')$ can only have been proved using using the rules update_rec and update_found. From the form of these rules we get $Id_1 = Id_2$.

- If we are in the update_rec case, then we also get $Id \neq Id_1$, $v_1 = v_2$, and necessarily

  $$(d_1 \overset{\text{update}}{\vdash} Id\backslash V \rightarrow d_2)$$

  In this case, the `lookup` statement cannot have been proved using the lookup_found rule. Then this statement has been proved using the lookup_rec rule and we have necessarily the statement

  $$(d_2 \overset{\text{lookup}}{\vdash} Id \mapsto V')$$

  Thus, we get the result from the induction hypothesis.

- If we are in the update_found case, then we get $Id_1 = Id_2 = Id$, $V = v_2$. and the lookup statement has necessarily be proved using the lookup_found rule. Thus we also get $v_2 = V'$, which gives the result.

3.2.2 Remarks on the inversion technique

When a relation R is defined using inference rules, we do not only intend that this relation is provable using these inference rules, but also that it is defined by these rules only. For this reason, when considering a statement of the form $R(t)$ where we know enough about t to decide which rule has been employed to prove this statement, we manage to assert that the premise of this rule is also true.

This is an inversion phenomenon. Let us consider the following rule.

$$\frac{A}{B}$$

Although this rule means $A \Rightarrow B$, we manage to use it as if it meant $B \Rightarrow A$. This phenomenon has been studied in detail by C. Cornes and D. Terrasse [4] and has led to proof tools that we will use later in this paper.

3.2.3 Proof by induction on proofs

In this section, we use the rules of figure 1 in section 2.2. Let us assume that we know a proof that the execution of two statements I and I' is equivalent, that is:

$$\forall D, D'. \quad D \vdash I \to D' \quad \Leftrightarrow D \vdash I' \to D'$$

Now we want to prove that for any expression E the execution of the two statements $\mathsf{while}(E, I)$ and $\mathsf{while}(E, I')$ are equivalent. Without loss of generality, it is enough to prove that the execution of one of the statements implies the execution of the other:

$$\forall D, D'. \quad D \vdash \mathsf{while}(E, I) \to D' \quad \Rightarrow D \vdash \mathsf{while}(E, I') \to D'$$

We will perform this proof by induction on the the proof of the statement $D \vdash \mathsf{while}(E, I') \to D'$. The proposition we want to prove by induction is as follows:

$$\forall d, i, d'. \quad d \vdash i \to d' \quad \Rightarrow i = \mathsf{while}(E, I) \Rightarrow d \vdash \mathsf{while}(E, I') \to d'$$

Base cases. The base cases correspond to the rules of the definition that do not have the relation $. \vdash . \to .$ among their premises. In the definition there are only three such rules. Two rules correspond to the statement being an empty sequence or an assignement. These cases do not apply here because they contradict the equality given as hypothesis. the third case is the while statement, when the expression evaluates to false (rule exec_while_false). In that case one has necessarily $D = D'$ and $D \vdash E \mapsto \mathsf{false}$, so one can obviously prove $D \vdash \mathit{while}(E, I') \to D'$, using the rule exec_while_false.

Induction cases. The Induction cases correspond to all the rules where a statement $d \vdash i \to d'$ appears. Among all these rules, only one applies here: the rule exec_while_true for the while statement when the expression evaluates to true. In this case, we have two premises of the form $d \vdash i \to d'$. The first has the following statement:

$$D \vdash I \to D'.$$

The second has the following statement:

$$D' \vdash \mathsf{while}(E, I) \to D''.$$

The induction hypothesis for the first of these premises has the following statement:

$$I = \mathsf{while}(E, I) \Rightarrow D \vdash \mathsf{while}(E, I') \to D'.$$

The induction hypothesis for the second of these premises has the following statement:

$$\mathsf{while}(E, I) = \mathsf{while(E, I)} \Rightarrow D'\mathsf{while}(E, I') \rightarrow D''$$

The first of these induction hypotheses is trivially useless since it needs a pre-condition that is not provable. On the other hand, the pre-condition of the second induction hypothesis is trivially true. Using this result, the fact that I and I' are equivalent, and the fact that E evaluates to true in the environment D, we can gather all the premises needed to apply the rule exec_while_true and construct a proof that $\mathsf{while}(E, I')$ can be executed in environment D and that it returns the environment D''.

4 Using a mechanized proof assistant

Proofs on formal specification are very formal and contain a lot of tedious details. It is reasonable to look for tools that will help mechanizing such tasks. We have experimented with the coq system [7]. This system has a powerfool treatment of inductive structures, so that it will support quite well the various techniques of proof by induction. So far the integration of our formal specifications with the Coq system provides the following features:

- Abstract syntax specifications and semantics specifications can be compiled into data-type declarations and axioms for the Coq system. It makes it possible to use the computer to check formal proofs of correctness regarding our specifications. In this respect it is a tool that provides an absolute criterion for verifying the absence of errors in specifications.
- Programming tools as found in our programming environment generator can improve the usability of the Coq system and make some proofs easier.

In this section, we are going to present the basic correspondence between formal specifications and Coq definitions. This will later be used to prove the correctness tools.

4.1 Using Coq

The type theory provided by the Coq system is a good tool to represent programming languages. The real gain of using this system is in the specialized proof methods provided for manipulating inductive data types.

4.1.1 Translating Specifications

Here we show the kind of Coq definitions that we use to represent languages and natural semantics specifications.

Translating the Abstract Syntax

As far as proofs are concerned, programming languages are free algebras. The word *algebra* means that a language is stable for a certain number of n-ary operations (the operators of the language), the word *free* indicates that two elements are equal if, and only if, these elements have been obtained in the same way from the basic operations (the same operators occur at the same positions). In Coq, such algebras are called inductive sets.

The correspondance is not completely straightforward because we actually consider *many-sorted* algebras. This simply expresses that we have phylum constraints in the construction of trees. The approach taken in this paper is to represent each phylum by a different inductive set, sometimes using mutually inductive sets. The presence of an operator in several phyla is taken care of by coercion operators.

To illustrate the translation of abstract syntax specifications towards Coq data structures and to avoid obfuscating the presentation, we use a toy language, dummy, described by the following abstract syntax:

```
  leaf -> implemented as SINGLETON;
  d_list -> A *...;
  binary -> B B ;
  L ::= d_list;
  A ::= L B;
  B ::= binary leaf;
```

This language description is translated into several Coq definitions.

```
Mutual [A:Set] Inductive
   l_list: Set :=
        l_nil: (l_list A)
       | l_cons: A -> (l_list A) ->(l_list A).

Mutual [A:Set] Inductive
   l_listp: Set :=
        l_nilp: A ->(l_listp A)
       | l_consp: A -> (l_listp A) ->(l_listp A).

Mutual Inductive
   B: Set :=
        binary: B -> B ->B
       | leaf: B.

Mutual Inductive
   A: Set :=
        coer_B_A: B ->A
       | coer_L_A: L ->A
   with
   L: Set :=
        d_list: (l_list A) ->L.
```

The phyla `A` and `B` are directly represented by the inductive sets `A` and `B`. and the operators `binary`, `leaf` and `d_list` are directly represented by the constructors with the same names. The main differences are in the treatement of lists, where polymorphic constructors are introduced (`l_listp` is actually not used, it would be if there were an operator of arity + in the language).

Translating phyla into properties

The solution of translating each phylum into a distinct inductive set is not the only solution. It has a few drawbacks, which we already pointed out when explaining how abstract syntax definitions could be translated into a formalism with no phylum inclusion. Also there is a need that the relation of inclusion between phyla be complete in some sense: if two phyla `A` and `B` have a non-empty intersection, the intersection between these phyla must be a phylum, and there should be multiple paths of inclusions: $C_1 \subset C_2 \subset C_3$ and $C_1 \subset C_4 \subset C_3$, with $C_2 \neq C_4$ should not happen.

There exists another encoding, which is more general. In this encoding the whole language is considered as one single inductive set. The phyla are then distinguished by properties, defined recursively over trees. The tools to define these properties are more flexible than the inductive sets presented in the solution above. The paper [13] contains an interesting discussion of these solutions and their implementations.

Translating Semantic Specifications

Semantic specifications are translated into collections of axioms grouped in inductive definitions. Each inference rule is compiled into a formula that is universally quantified over the variables that occur in the rule.

To illustrate this translation, we take the Typol specification of a property silly defined on elements of our language dummy:

```
set silly is
judgement ⊢ dummy;

found: ⊢ binary(leaf(), leaf()) ;

list_rec1:
  ⊢ l_1
  ---------------------
  ⊢ d_list[l_1. l_2] ;

list_rec2:
  ⊢ l_2
  ---------------------
  ⊢ d_list[l_1. l_2] ;
```

Before continuing, we should explain a few notions of Coq syntax. The statement $\forall x : A.\ B$ reads as "for all x of type A, B is true". The statement $A \Rightarrow B$ reads alternatively as the type of functions from A to B when A and B are sets (as we already saw when translating abstract syntaxes) or as the sentence "A implies B". Also, $A \Rightarrow B \Rightarrow C$ is implicitly parenthesized around $B \Rightarrow C$ and it reads alternatively as the type of function that take an argument of type A and an argument of type B and return an argument of type C or as the sentence "A and B imply C".

Thus, the rule that describes the search in the first element of a list is translated into the axiom list_rec1 given below:

```
list_rec1:
    ∀ l_2:( l_list  A). ∀ l_1: A.
       ( silly l_1) ⇒
         ( silly ( coer_L_A ( d_list ( l_cons  A l_1 l_2))))
```

In this formula the translation of the judgement $\vdash l_1$ from the Typol specification is the application of the relation silly to an argument. Note that the sign $\Rightarrow$ is used here to represent implications.

Coq also provides functionalities to define relations inductively. Given a collection of statements about a relation, an inductive definition expresses that the defined relation is the *least* relation, in terms of set inclusion, that verifies all these statements. Actually, semantic specifications correspond to inductive definitions. For instance in the case of our silly property, one should be able to express that the term leaf does not verify this property, because there is no way to construct a statement silly(leaf) using only the axioms found, list_rec1, and list_rec2. The right way to translate the specification of silly is thus to write down the following inductive definition:

```
Mutual Inductive
  silly : A ⇒ Prop :=
      found: (silly (coer_B_A (binary leaf leaf)))
    | list_rec1:
        ∀l_2:(l_list A).∀l_1:A.(silly l_1) ⇒
        (silly (coer_L_A (d_list (l_cons A l_1 l_2))))
    | list_rec2:
        ∀l_1:A.∀l_2:(l_list A).(silly (coer_L_A (d_list l_2))) ⇒
        (silly (coer_L_A (d_list (l_cons A l_1 l_2)))).
```

For this inductive definition, the Coq system generates the following axiom, which expresses exactly that silly is the least property verifying axioms found, list_rec1, and list_rec2:

```
silly_ind
 :
 ∀P:A ⇒ Prop.
 (P (coer_B_A (binary leaf leaf))) ⇒
 (∀l_2:(l_list A).∀l_1:A.(silly l_1) ⇒ (P l_1) ⇒
  (P (coer_L_A (d_list (l_cons A l_1 l_2))))) ⇒
 (∀l_1:A.
  ∀l_2:(l_list A).
  (silly (coer_L_A (d_list l_2))) ⇒ (P (coer_L_A (d_list l_2))) ⇒
  (P (coer_L_A (d_list (l_cons A l_1 l_2))))) ⇒ ∀a:A.(silly a) ⇒ (P a)
```

This axiom expresses that for any property P defined on Dummy, such that

- P is true on (binary leaf leaf) (this premise corresponds to the rule found),
- if silly holds for l_1 and P is true for l_1 then it is true for (d_list (l_cons l_1 l_2)) (this premise corresponds to the rule list_rec1),
- if silly holds for l_2 P is true for l_2 then it is true for (d_list (l_cons l_1 l_2)) (this premise corresponds to the rule list_rec2),

then P defines a subset of Dummy that is greater than the one defined by silly: for every l in Dummy if silly holds for l then P does.

Thanks to the work of D. Terrasse [13], the relevant inductive definitions are automatically generated from Typol specifications. In turn, the induction theorem is automatically generated by the Coq system.

4.1.2 Proof Methods

The Coq system provides a whole set of functionalities around inductive definitions. Proof of properties of programs often rely on proofs by induction of various forms. This section describes these proof methods.

Structural Induction

The Coq system generates automatically a structural induction principle for each phylum. However, one has to be careful with the induction principles for types in a package of mutually inductive types. There, the `Scheme` command makes it possible to generate the right set of induction principles. In our case, the phyla `A` and `L` are mutually inductive, and we need to generate a powerful induction scheme:

```
Scheme L_A_ind := Induction for L Sort Prop
   with
   A_L_ind := Induction for A Sort Prop.
```

This yields two inductive principles with the following statements:

```
L_A_ind
 :
 ∀P:L ⇒Prop.
 ∀P0:A ⇒Prop.
 (∀l:(l_list A).(P (d_list l))) ⇒
 (∀b:B.(P0 (coer_B_A b))) ⇒(∀l:L.(P l) ⇒(P0 (coer_L_A l))) ⇒
  ∀l:L.(P l)
A_L_ind
 :
 ∀P:L ⇒Prop.
 ∀P0:A ⇒Prop.
 (∀l:(l_list A).(P (d_list l))) ⇒
 (∀b:B.(P0 (coer_B_A b))) ⇒(∀l:L.(P l) ⇒(P0 (coer_L_A l))) ⇒
  ∀a:A.(P0 a)
```

Moreover, the Coq system makes it possible to write recursive function by cases over trees in the language. The induction principle associated with each phyla and this possibility make it possible to state all the basic properties that express that a programming language is a free algebra.

5 Proving Transformations Correct

In the first sections of this paper, we have described two methods for manipulating programs. A first method was based on rewrite rules that the user could trigger by selecting an expression and a rule in a menu. The second method was based on specifications written in Typol. We are now going to study proofs that transformations written in these styles preserve the meaning of programs.

5.1 A simple Rewrite Rule

Let us consider the following rewrite rule:

```
'trivial if' : if(*exp, sequence(), sequence()) : {INST}
   --> sequence()
```

This rule replaces conditional instructions where both branches do nothing by an instruction that does nothing. It is clear intuitively that applying this rule on any subterm of a program will not change the behavior of the program. We want to check a proof of this property using the Coq system. The figure 2 contains a complete formalisation of this property.

A first step is to implement the rewrite rule as a Coq inductive definition: trivial_if_rel. We have to take into account the fact that the rewriting can occur on any subterm of the program which is an instruction and matches the lefthand side. Thus, the definition contains extra constructors to describe the recursive descent to the reduced expression.

We can now come to our main goal, to show that this rewriting relation preserves the meaning of programs. We have to state that the execution of a program and its transformed version from the same initial state terminate in the same conditions and yield the same output state. Actually, this is not completely true: it may be that the transformed program executes correctly while the original program would have failed to execute, due to errors present in the removed expression. So we will prove only a limited form of soundness, expressed by the theorem trivial_if_sound.

$$\mathsf{trivial\_if\_sound}: \quad (D \vdash I \to D' \quad \& \quad \mathsf{trivial\_if\_rel}(I, I')) \Rightarrow D \vdash I' \to D'$$

Note that the theorem trivial_if_sound is also present, in a different form, in the Coq script shown in figure 2. There, the execution judgement is represented by (exec d i d').

Described informally, the proof comes from the fact that because trivial_if_rel(i ,i') holds we can infer the existence of a derivation for $D \vdash I' \to D'$ from any derivation of $D \vdash I \to D'$. Thus, this proof is performed

```
Require Export little.
Require Export dynamics.

Mutual Inductive
  trivial_if_rel: INST ⇒ INST ⇒ Prop :=
      tir_if_1:
        ∀e:EXP. ∀i1, i2, i'1:INST. (trivial_if_rel i1 i'1) ⇒
        (trivial_if_rel (l_if e i1 i2) (l_if e i'1 i2))
    | tir_if_2:
        ∀e:EXP. ∀i1, i2, i'2:INST. (trivial_if_rel i2 i'2) ⇒
        (trivial_if_rel (l_if e i1 i2) (l_if e i1 i'2))
    | tir_while:
        ∀e:EXP. ∀i, i':INST. (trivial_if_rel i i') ⇒
        (trivial_if_rel (while e i) (while e i'))
    | tir_sequence_1:
        ∀i, i':INST. ∀l:(l_list INST). (trivial_if_rel i i') ⇒
        (trivial_if_rel
            (coer_SEQUENCE_INST (sequence (l_cons INST i l)))
            (coer_SEQUENCE_INST (sequence (l_cons INST i' l))))
    | tir_sequence_2:
        ∀i:INST.
        ∀l, l':(l_list INST).
        (trivial_if_rel
           (coer_SEQUENCE_INST (sequence l))
           (coer_SEQUENCE_INST (sequence l'))) ⇒
        (trivial_if_rel
            (coer_SEQUENCE_INST (sequence (l_cons INST i l)))
            (coer_SEQUENCE_INST (sequence (l_cons INST i l'))))
    | trivial_if:
        ∀e:EXP.
        (trivial_if_rel
            (l_if
               e (coer_SEQUENCE_INST (sequence (l_nil INST)))
               (coer_SEQUENCE_INST (sequence (l_nil INST))))
            (coer_SEQUENCE_INST (sequence (l_nil INST)))).
Hint exec_assign exec_if_true exec_if_false exec_while_false
     exec_sequence_end exec_while_true exec_sequence_rec.

Theorem trivial_if_sound:
  ∀i:INST. ∀d, d':DECLS. (exec d i d') ⇒
  ∀i':INST. (trivial_if_rel i i') ⇒ (exec d i' d').
1: Intros i d d' H'; Elim H';
    Try (Intros;
          (Matchhyp Y with (λd, d':DECLS. λi:INST. (exec d i d')) Then
          (Generalize Y) End);
          (Matchhyp X with (λi, i':INST. (trivial_if_rel i i')) Then
          (Inversion X) End); EAuto; Exact fail).
The case of while − true remains, because it is a case that uses two distinct instance of
induction hypotheses.
1: Intros D' D'' D E I Heval Hprem1 Hrec1 Hprem2 Hrec2 i' Htriv_if;
    Inversion Htriv_if; Apply exec_while_true with D' := D'; Auto;
    (Matchhyp X with (λi:INST. (while E i) =i') Then (Rewrite X) End);
    Auto.
Qed.
```

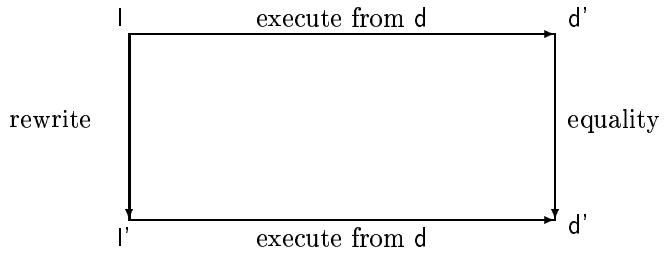
Figure 2: A Coq script to prove the soundness of `'trivial if'`

by an induction on the proof of $D \vdash I \to D'$. In the Coq script, this induction is requested by the command *Elim* H'. This induction yields seven case.

In all cases, we perform a second case analysis on the proof of trivial_if_rel(I, I'). In two of the cases of the induction proof, this case analysis shows that trivial_if_rel(I, I') is impossible for any I', these are the cases when I is an assignment or an empty sequence. Then in four cases (when I is a if statement, a sequence, or a while loop with the conditional expression evaluating to false), the second case analyses gives enough data to conclude trivially using one of the induction hypootheses. The last remaining case is the while loop, when the expression evaluates to true. Here again, we perform a case analysis on the proof of trivial_if_rel, but this case is a bit more complicated because the two induction hypotheses will be used to solve it.

let us remark a few facts about the Coq script for this proof. First the case analysis on the the proof of trivial_if_rel is performed using the command *Inversion* X or *Inversion* Htriv_if. This command performs the case analysis, solves automatically all the trivial cases (like the impossible cases of the assignment and the empty sequence), infers a collection of equalities and rewrites the goal with these equalities. It may happen that the rewriting is a bit clumsy: this is what happens in the non-trivial case for the while statement, where we follow the case analysis by an extra rewriting, that undoes one of those performed by the Inversion Htriv_if command. Second, we use a tactical that is not provided by the Coq system: the *Matchhyp* tactical. This tactical searches an assumption that maches the expression given in the *with* clause. When this assumption is found it binds its name to the identifier provided by the user and executes the command given in the *Then* clause. This command makes it possible to write a command that will apply to several cases produced in a proof by induction, even though all these cases have different forms : different number of premises, different number of variables, different numbers of induction hypotheses.

Let us conclude with a few remarks on the form of the proof we just performed, since some aspects will recur again and again when working on programming language semantics. First of all, this proof can be schematically described by a commutative diagram. We have an initial piece of data, an instruction, that we can process in two different ways. In this case, we can either execute this instruction, which yields a result environment, or rewrite it, which yields a new instruction. The question we have solved is whether the two outputs can be reunited in some way. Put in another way, the question is whether executing the rewritten instruction yields a result that is equal to the output of executing the initial instruction. So we have a diagram with the following shape:



Here the diagram is somehow degenerated, as the rightmost arrow is only an equality. However it happens that this arrow corresponds to a non trivial correspondence. The reading of this diagram is quite simple: for every possible way of realizing the top and left arrows, one has to prove that there exist a value for the bottom right corner of the diagram such that the bottom and right arrows also exist. If this can be proved, one says that the diagram commutes. This kind of proofs has already been done by hand very often, like for example in [5].

5.2 Proving a Natural Semantics Specification

We now come to the exercise of proving that a transformation specified in natural semantics is coherent with the dynamic semantics of the language. The basic principles are the same: we are going to show that some commutative diagrams can be closed, and we are going to use inductions on the size of derivations

together with case analyses on rules. Here, however, the proof will be more complex because the constant propagation transformation uses its own notion of environment that we shall have to correlate with the notion of environment used in the operational semantics. We are not going to indicate the detail of the proof but the organization in lemmas. All the lemmas are given in annex F.

a first collection of lemmas studies the properties of the environment used in dynamic semantics itself, and the relations used to manipulate these environments: update, bound. These lemmas are the lemmas: update_diff, update_diff_rev, update_same, update_commute, update_commute2, update_twice, update_trans.

The constant propagation specification uses sequences of assignment to represent known information about constants. Obviously, any sequence of assignments does not fit for this purpose, and many statements will be using the fact that the sequence use as list of binding is well-formed in some sense. Thus, the proof introduces a predicate wf_binding (meaning *well formed binding*) to describe the bindings that are expected.

An important collection of lemmas shows that the relations partial_update, propag_bound, get_binding, glb and remove that manipulate directly these sequences preserve their "well formedness": get_binding_wf, glb1_wf, glb_wf1,glb2_wf, glb_wf2, remove_wf, partial_update_wf, propagation_wf.

A collection of lemmas studies the pivotal importance of the property that an identifier can be absent from a list of binding, and the interaction of this property with the primitives that manipulate bindings: glb1notin, get_binding_not_not_in, get_binding_not_in_trans, glb2notin, glb_not_in_transmit1, glb_not_in_transmit2, get_binding_not_in_both, remove_not_in, not_in_remove, not_in_propag_bound, partial_update_not_in.

Since lists of bindings are sequences of assignments, they can also be used to act on dynamic semantics environments, simply by executing these bindings. A collection of lemmas studies the way the not_in property acts in this context: compatible_untouched, compatible_untouched_rev, update_not_in, update_not_in_rev.

A collection of lemmas considers the combinations of primitive operations on bindings: partial_update, get_binding, propag_bound, and remove: propag_bound_remove_diff, propag_bound_remove_eq.

Last, the most important collection of lemmas compares the principal relation with their counterpart in the dynamic semantics: propag_bound with respect to bound, propag_eval with respect to eval, partial_update with respect to update, and propagation with respect to exec. These lemmas are the lemmas with "correct" in their name.

Note that we did not prove completely the correctness of the constant propagation information. The statement of propagation_correct1 only indicates that if I' is the result of the transformation on I, and if I can be executed then it will be possible to execute I' followed by the final sequence of assignments to obtain the same result, in some sense. This statement does not indicate that all executions of I' will return the right value. This statement can be taken care of by a proof that the dynamic semantics is deterministic. But this statement also does not say anything about the case where I does not terminate. It might happen that I' terminates when I does not. There is a need for another lemma where the fact that I' terminates is taken as an hypothesis.

6 Conclusion

The work we have described in these notes has two facets. First, we have described the various documents that can be produced to formally specify a programming language and we have shown how these documents could be used to implement tools adapted to this language. Then, we have shown that these specifications could also be "compiled" into input data for a proof system and we have describe a few proof techniques commonly used to reason about programming languages. Thus, these notes have actually proposed a complete range of tools for the researcher interested in defining a new language, experimenting with tools around this language, and proving some of its properties.

From a designer's point of view however, this work also exposes the current weaknesses of the environment. It is unfortunate that the objects manipulated while proving statements about the language are so different from the object manipulated while designing the language. Also, it often seems that the user has to provide

the proof of too many trivial facts. This feature makes the task of proving properties of a program or a programming language very time consuming. For instance, only proving the correctness of the constant propagation on expressions took the author more than a month of work, compared with a few days of work to design the programming environment tools.

References

[1] Isabelle Attali. Compiling typol with attribute grammars. In *International Worksop on Programming Language Implementation and Logic Programming*. Springer Verlag LNCS, May 1988.

[2] Yves Bertot and Ranan Fraer. Reasonning with Executable Specifications. In *TAPSOFT'95*, volume 915, pages 531–545, 1995. Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT'95).

[3] Debra Cameron and Bill Rosenblatt. *Learning GNU Emacs*. O'Reilly & Associates, Inc., 1991.

[4] Christina Cornes and Delphine Terrasse. Automatizing inversion of inductive predicates in coq. In *Types for Proofs and Programs*, volume 1158 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[5] Joëlle Despeyroux. Proof of translation in natural semantics. In *Symposium on Logic in Computer Science*, pages 193–205. IEEE Computer Society, June 1986.

[6] Joëlle Despeyroux and André Hirshowitz. Higher-order syntax with induction in Coq. In *Fifth International Conference on Logic Programming and Automated Reasonning*, 1994.

[7] INRIA. *The Coq Proof Assistant Reference Manual*, December 1996. Version6.1.

[8] Ian Jacobs and Janet Bertot, editors. *Centaur 1.2*, chapter The PPML Manual. Inria Sophia–Antipolis, 1993.

[9] Ian Jacobs and Janet Bertot, editors. *Centaur 1.2*, chapter The Typol Manual. Inria Sophia–Antipolis, 1993.

[10] Gordon Plotkin. Structural operational semantics. lecture notes DAIMI FN-19, Aarhus University, 1981. (reprinted 1991).

[11] Thomas Reps. Optimal-time incremental semantic analysis for syntax-directed editors. In *Ninth ACM Conference on Principles of Programming Languages*, 1982.

[12] Thomas Reps and Tim Teitelbaum. *The Synthesizer Generator: a system for constructing language based editors*. Springer Verlag, 1988. (third edition).

[13] D. Terrasse. Encoding natural semantics in coq. In *Proceedings of the Fourth International Conference on Algebraic Methodology and Software Technology, AMAST'95*, Springer-Verlag LNCS, July 1995.

[14] Glynn Winskel. *The Formal Semantics of Programming Languages, an introduction*. Foundations of Computing. The MIT Press, 1993.

Appendices

These appendices contain the various specifications for the language little:

- The first description gives the abstract syntax and the concrete syntax,
- the second description gives the formatting specification,
- the third description gives the dynamic semantics,
- the fourth description gives the specification of the constant propagation tool.

We also provide a few files related to the proofs performed about the language:

- the translation of the dynamic semantics,
- all the lemmas involved in the proof of correctness of constant propagation.

A Syntax specification of `little`

```
definition of little is
   chapter 'plain syntax'
      abstract syntax
         PROGRAM  ::= program ;
         program  -> DECLS INST ;
         DECLS  ::= decls ;
         decls  -> DECL * ... ;
         DECL  ::= decl ;
         decl  -> ID TYPE VAL ;
         TYPE  ::= boolean integer ;
         VAL  ::= INT BOOL ;
         BOOL  ::= true false ;
         true  -> implemented as SINGLETON ;
         false  -> implemented as SINGLETON ;
         boolean  -> implemented as SINGLETON ;
         integer  -> implemented as SINGLETON ;
         INT  ::= int ;
         int  -> implemented as INTEGER ;
         INST  ::= assign sequence if while ;
         assign  -> ID EXP ;
         if  -> EXP INST INST ;
         while  -> EXP INST ;
         sequence  -> INST * ... ;
         SEQUENCE  ::= sequence ;
         EXP  ::= VAL ID and or gt eq plus minus not ;
         and  -> EXP EXP ;
         or  -> EXP EXP ;
         gt  -> EXP EXP ;
         eq  -> EXP EXP ;
         plus  -> EXP EXP ;
         minus  -> EXP EXP ;
         not  -> EXP ;
         ID  ::= ident ;
         ident  -> implemented as IDENTIFIER ;
```

```
rules
   <program> ::= "declarations" <decls> "in" <inst> ;
      program (<decls>, <inst>)
   <decls> ::= <decls> ";" <decl> ;
      decls-post (<decls>, <decl>)
   <decls> ::= <decl> ;
      decls-list ((<decl>))
   <decl> ::= "variable" <ident> ":" <type> "=" <value> ;
      decl (<ident>, <type>, <value>)
   <type> ::= "integer" ;
      integer ()
   <type> ::= "boolean" ;
      boolean ()
   <ident> ::= %IDENT ;
      ident-atom(%IDENT)
   <value> ::= <int> ;
      <int>
   <int> ::= %INT ;
      int-atom(%INT)
   <value> ::= "true" ;
      true ()
   <value> ::= "false" ;
      false ()
   <inst> ::= "begin" <sequence> "end" ;
      <sequence>
   <sequence> ::= <sequence> <inst> ";" ;
      sequence-post (<sequence>, <inst>)
   <sequence> ::= ;
      sequence-list (())
   <inst> ::= <ident> ":=" <exp> ;
      assign (<ident>, <exp>)
   <inst> ::= "if" <exp> "then" <inst> "else" <inst> ;
      if (<exp>, <inst>.1, <inst>.2)
   <inst> ::= "while" <exp> "do" <inst> ;
      while (<exp>, <inst>)
   <exp> ::= <exp> "and" <exp1> ;
      and (<exp>, <exp1>)
   <exp> ::= <exp> "or" <exp1> ;
      or (<exp>, <exp1>)
   <exp> ::= <exp1> ;
      <exp1>
   <exp1> ::= <exp2> ">" <exp2> ;
      gt (<exp2>.1, <exp2>.2)
   <exp1> ::= <exp2> "=" <exp2> ;
      eq (<exp2>.1, <exp2>.2)
   <exp1> ::= <exp2> ;
      <exp2>
   <exp2> ::= <exp2> "+" <exp3> ;
      plus (<exp2>, <exp3>)
   <exp2> ::= <exp2> "-" <exp3> ;
      minus (<exp2>, <exp3>)
   <exp2> ::= <exp3> ;
```

```
            <exp3>
         <exp3> ::= "not" "(" <exp> ")" ;
            not (<exp>)
         <exp3> ::= <ident> ;
            <ident>
         <exp3> ::= <value> ;
            <value>
         <exp3> ::= "(" <exp> ")" ;
            <exp>
   end chapter ;

   chapter 'entry points'
      rules
         <program> ::= "[ID]" <ident> ;
            <ident>
         <program> ::= "[VAL]" <value> ;
            <value>
         <program> ::= "[DECL]" <decl> ;
            <decl>
         <program> ::= "[DECLS]" <decls> ;
            <decls>
         <program> ::= "[INST]" <inst> ;
            <inst>
         <program> ::= "[EXP]" <exp> ;
            <exp>
         <program> ::= "[TYPE]" <type> ;
            <type>
   end chapter ;
end definition
```

B Formatting specification

```
--  $Id: little-std.ppml,v 1.1 96/03/22 11:27:43 bertot Exp $
prettyprinter std of little is

default
    <h 1>;
    <v 0,0>;
    <hv 1,0,0>;
    <hov 1,0,0>;

function precedence is
    precedence (ident)= 0;
    precedence (int)= 0;
    precedence (true)= 0;
    precedence (false)= 0;
    precedence (#not)= 0;
    precedence (#and(*e1, *e2))= 6;
    precedence (#or(*e1, *e2))= 5;
    precedence (gt(*e1, *e2))= 4;
    precedence (eq(*e1, *e2))= 3;
    precedence (plus(*e1, *e2))= 2;
    precedence (minus(*e1, *e2))= 1;
    precedence (*x)= 7;
end precedence ;

rules

*x ^edit -> [<v> in class = edit : [<h> ^edit]];

program(*d, *i) ->
    [<v> in class = keyword : "declarations" *d in class = keyword : "in" *i];

decls(*first, **x) -> [<v> *first (<h 0> ";" **x)];

decl(*id, *type, *val) ->
    [<hv> in class = keyword : "variable" *id ":" *type "=" *val];

true -> [<h> "true"];

integer -> [<h> "integer"];

boolean -> [<h> "boolean"];

false -> [<h> "false"];

assign(*id, *exp) -> [<hv 1,0,0> [<h 1> *id ":="] *exp];

#if(*e, *i1, *i2) ->
    [<hov>
     [<hv> in class = keyword : "if" *e]
     [<hov 1,1,0> in class = keyword : "then" *i1]
```

```
     [<hov 1,1,0> in class = keyword : "else" *i2]];

while(*e, *i) ->
    [<hov 1,2,0>
     [<hv> in class = keyword : "while" *e in class = keyword : "do"] *i];

sequence(**x) ->
    [<v>
     [<v 1,0> in class = keyword : "begin" [<v 0,0> (**x <h 0> ";")]]
     in class = keyword : "end"];

*x where
 *x in
  {#and(*e1, *e2), #or(*e1, *e2), plus(*e1, *e2), minus(*e1, *e2), gt(*e1, *e2),
  eq(*e1, *e2)} -> [<hv> if precedence(*e1) <= precedence(*x) then *e1
                         else parentheses::*e1
                         end if
                         operator::*x
                         if precedence(*e2) < precedence(*x) then *e2
                         else parentheses::*e2
                         end if];

#not(*e) ->
    [<h>
     in class = keyword : "not" in class = keyword : "(" *e
     in class = keyword : ")"];

operator::#and(*e1, *e2) -> [<h> in class = keyword : "and"];

operator::#or(*e1, *e2) -> [<h> in class = keyword : "or"];

operator::plus(*e1, *e2) -> [<h> "+"];

operator::minus(*e1, *e2) -> [<h> "-"];

operator::eq(*e1, *e2) -> [<h> "="];

operator::gt(*e1, *e2) -> [<h> ">"];

parentheses::*x -> [<h 0> "(" *x ")"];

end prettyprinter
```

C Dynamic semantics

C.1 Evaluating expressions

```
--  $Id: eval.ty,v 1.1 96/03/21 17:19:56 bertot Exp $
program eval is
use little;
import
   id_diff(ID, ID), val_diff(VAL, VAL), op_and(VAL, VAL, VAL),
   op_or(VAL, VAL, VAL), op_gt(VAL, VAL, VAL), op_plus(VAL, VAL, VAL),
   op_minus(VAL, VAL, VAL), op_eq(VAL, VAL, VAL), op_not(VAL, VAL) from pl_misc;

   set eval is
   judgement DECLS |- EXP |-> VAL;

   eval_true: D |- true() |-> true() ;

   eval_false: D |- false() |-> false() ;

   eval_int: D |- int N |-> int N ;

   eval_ident:
      bound(D |- ident N |-> V)
      ----------------
      D |- ident N |-> V ;

   interpr_and:
      D |- E1 |-> V1  &  D |- E2 |-> V2  &  op_and(V1, V2, V)
      ----------------
      D |- and(E1, E2) |-> V ;

   interpr_or:
      D |- E1 |-> V1  &  D |- E2 |-> V2  &  op_or(V1, V2, V)
      ----------------
      D |- or(E1, E2) |-> V ;

   interpr_gt:
      D |- E1 |-> V1  &  D |- E2 |-> V2  &  op_gt(V1, V2, V)
      ----------------
      D |- gt(E1, E2) |-> V ;

   interpr_plus:
      D |- E1 |-> V1  &  D |- E2 |-> V2  &  op_plus(V1, V2, V)
      ----------------
      D |- plus(E1, E2) |-> V ;

   interpr_minus:
      D |- E1 |-> V1  &  D |- E2 |-> V2  &  op_minus(V1, V2, V)
      ----------------
      D |- minus(E1, E2) |-> V ;

   interpr_eq:
```

```
      D |- E1 |-> V1  &  D |- E2 |-> V2  &  op_eq(V1, V2, V)
      ----------------
      D |- eq(E1, E2) |-> V ;

   interpr_not:
      D |- E1 |-> V1  &  op_not(V1, V)
      ----------------
      D |- not(E1) |-> V ;
   end eval;

   set bound is
   judgement DECLS |- ID |-> VAL;

   lookup_found: decls[decl(I, _, V).D] |- I |-> V ;

   lookup_rec:
      D |- I |-> V
      ----------------
      decls[decl(I', _, V').D] |- I |-> V ;
               provided id_diff(I, I');
   end bound;
end eval;
```

C.2 Updating the environment

```
--  $Id: update.ty,v 1.1 96/03/21 17:19:59 bertot Exp $
program update is
use little;
import id_diff(ID, ID) from pl_misc;

   set update is
   judgement DECLS |- ID \ VAL -> DECLS;

   update_exact: decls[decl(I, T, _).D] |- I \ V -> decls[decl(I, T, V).D] ;

   update_rec:
      D |- I \ V -> D'
      ----------------
      decls[decl(I', T, V').D] |- I \ V -> decls[decl(I', T, V').D'] ;
               provided id_diff(I, I');
   end update;
end update;
```

C.3 Executing instructions: main specification

```
--  $Id: dynamics.ty,v 1.2 96/03/21 17:24:53 bertot Exp $
program dynamics is
use little;
import eval(DECLS |- EXP |-> VAL) from eval;
import update(DECLS |- ID \ VAL -> DECLS) from update;
```

```
export dynamics(|- P -> D) as execute(P) = D ;

   set dynamics is
   judgement |- PROGRAM -> DECLS;

   start_program:
      exec(D |- I -> D')
      ----------------
      |- #program(D, I) -> D' ;
   end dynamics;

   set exec is
   judgement DECLS |- INST -> DECLS;

   exec_assign:
      eval(D |- E |-> V)  &  update(D |- Id \ V -> D')
      ----------------
      D |- assign(Id, E) -> D' ;

   exec_if_true:
      eval(D |- E |-> true())  &  D |- I1 -> D'
      ----------------
      D |- if(E, I1, I2) -> D' ;

   exec_if_false:
      eval(D |- E |-> false())  &  D |- I2 -> D'
      ----------------
      D |- if(E, I1, I2) -> D' ;

   exec_while_false:
      eval(D |- E |-> false())
      ----------------
      D |- while(E, I) -> D ;

   exec_while_true:
      eval(D |- E |-> true())  &  D |- I -> D'  &  D' |- while(E, I) -> D''
      ----------------
      D |- while(E, I) -> D'' ;

   exec_sequence_end: D |- sequence[] -> D ;

   exec_sequence_rec:
      D |- I1 -> D'  &  D' |- I2 -> D''
      ----------------
      D |- sequence[I1.I2] -> D'' ;
   end exec;
end dynamics;
```

D Program transformation

D.1 Combining two lists of constants

```
program glb is
use little;
import val_diff(VAL, VAL), id_diff(ID, ID) from pl_misc;

   set glb is
   judgement (SEQUENCE + SEQUENCE = SEQUENCE + SEQUENCE + SEQUENCE);

   glb_equal:
      get_binding(D2 = assign(I, V) + D2')  &  (D1 + D2' = D' + E1 + E2)
      ----------------
      (sequence[assign(I, V).D1] + D2 = sequence[assign(I, V).D'] + E1 + E2) ;

   glb_diff:
      get_binding(D2 = assign(I, V') + D2')  &  (D1 + D2' = D' + E1 + E2)
      ----------------
      (sequence[assign(I, V).D1] +
          D2 = D' + sequence[assign(I, V).E1] + sequence[assign(I, V').E2]) ;
               provided val_diff(V, V');

   glb_not_in:
      not_in(I, D2)  &  (D1 + D2 = D' + E1 + E2)
      ----------------
      (sequence[assign(I, V).D1] + D2 = D' + sequence[assign(I, V).E1] + E2) ;

   glb_end: (sequence[] + D2 = sequence[] + sequence[] + D2) ;
   end glb;

   set get_binding is
   judgement (SEQUENCE = INST + SEQUENCE);

   get_binding_found: (sequence[assign(I, V).D] = assign(I, V) + D) ;

   get_binding_rec:
      (D = assign(I, V) + D')
      ----------------
      (sequence[assign(I', V').D] = assign(I, V) + sequence[assign(I', V').D']) ;
               provided id_diff(I, I');
   end get_binding;

   set not_in is
   judgement (ID, SEQUENCE);

   not_in_end: (I, sequence[]) ;

   not_in_rec:
      (I, D)
      ----------------
      (I, sequence[assign(I', V).D]) ;
```

```
              provided id_diff(I, I');
   end not_in;
end glb;
```

D.2 Propagating constants through expressions

```
program propag_eval is
use little;
import
   id_diff(ID, ID), not_val(EXP), op_and(VAL, VAL, VAL), op_or(VAL, VAL, VAL),
   op_gt(VAL, VAL, VAL), op_plus(VAL, VAL, VAL), op_minus(VAL, VAL, VAL),
   op_not(VAL, VAL), op_eq(VAL, VAL, VAL) from pl_misc;

   set propag_eval is
   judgement SEQUENCE |- EXP |-> EXP;
   var val1, val2: VAL;

   propag_eval_true: D |- true() |-> true() ;

   propag_eval_false: D |- false() |-> false() ;

   propag_eval_int: D |- int N |-> int N ;

   propag_eval_ident:
      propag_bound(D |- ident N |-> V)
      ----------------
      D |- ident N |-> V ;

   propag_eval_and_val:
      D |- E1 |-> val1  &  D |- E2 |-> val2  &  op_and(val1, val2, V)
      ----------------
      D |- and(E1, E2) |-> V ;

   propag_eval_and_not_val1:
      D |- E1 |-> V1  &  D |- E2 |-> V2
      ----------------
      D |- and(E1, E2) |-> and(V1, V2) ;
               provided not_val(V1);

   propag_eval_and_not_val2:
      D |- E1 |-> V1  &  D |- E2 |-> V2
      ----------------
      D |- and(E1, E2) |-> and(V1, V2) ;
               provided not_val(V2);

   propag_eval_or_val:
      D |- E1 |-> val1  &  D |- E2 |-> val2  &  op_or(val1, val2, V)
      ----------------
      D |- or(E1, E2) |-> V ;

   propag_eval_or_not_val1:
      D |- E1 |-> V1  &  D |- E2 |-> V2
```

```
   ----------------
   D |- or(E1, E2) |-> or(V1, V2) ;
            provided not_val(V1);

propag_eval_or_not_val2:
   D |- E1 |-> V1  &  D |- E2 |-> V2
   ----------------
   D |- or(E1, E2) |-> or(V1, V2) ;
            provided not_val(V2);

propag_eval_gt_val:
   D |- E1 |-> val1  &  D |- E2 |-> val2  &  op_gt(val1, val2, V)
   ----------------
   D |- gt(E1, E2) |-> V ;

propag_eval_gt_not_val1:
   D |- E1 |-> V1  &  D |- E2 |-> V2
   ----------------
   D |- gt(E1, E2) |-> gt(V1, V2) ;
            provided not_val(V1);

propag_eval_gt_not_val2:
   D |- E1 |-> V1  &  D |- E2 |-> V2
   ----------------
   D |- gt(E1, E2) |-> gt(V1, V2) ;
            provided not_val(V2);

propag_eval_plus_val:
   D |- E1 |-> val1  &  D |- E2 |-> val2  &  op_plus(val1, val2, V)
   ----------------
   D |- plus(E1, E2) |-> V ;

propag_eval_plus_not_val1:
   D |- E1 |-> V1  &  D |- E2 |-> V2
   ----------------
   D |- plus(E1, E2) |-> plus(V1, V2) ;
            provided not_val(V1);

propag_eval_plus_not_val2:
   D |- E1 |-> V1  &  D |- E2 |-> V2
   ----------------
   D |- plus(E1, E2) |-> plus(V1, V2) ;
            provided not_val(V2);

propag_eval_minus_val:
   D |- E1 |-> val1  &  D |- E2 |-> val2  &  op_minus(val1, val2, V)
   ----------------
   D |- minus(E1, E2) |-> V ;

propag_eval_minus_not_val1:
   D |- E1 |-> V1  &  D |- E2 |-> V2
   ----------------
```

```
      D |- minus(E1, E2) |-> minus(V1, V2) ;
               provided not_val(V1);

   propag_eval_minus_not_val2:
      D |- E1 |-> V1  &  D |- E2 |-> V2
      ----------------
      D |- minus(E1, E2) |-> minus(V1, V2) ;
               provided not_val(V2);

   propag_eval_eq_val:
      D |- E1 |-> val1  &  D |- E2 |-> val2  &  op_eq(val1, val2, V)
      ----------------
      D |- eq(E1, E2) |-> V ;

   propag_eval_eq_not_val1:
      D |- E1 |-> V1  &  D |- E2 |-> V2
      ----------------
      D |- eq(E1, E2) |-> eq(V1, V2) ;
               provided not_val(V1);

   propag_eval_eq_not_val2:
      D |- E1 |-> V1  &  D |- E2 |-> V2
      ----------------
      D |- eq(E1, E2) |-> eq(V1, V2) ;
               provided not_val(V2);

   propag_eval_not_val:
      D |- E1 |-> val1  &  op_not(val1, V)
      ----------------
      D |- not(E1) |-> V ;

   propag_eval_not_not_val1:
      D |- E1 |-> V1
      ----------------
      D |- not(E1) |-> not(V1) ;
               provided not_val(V1);
   end propag_eval;

   set propag_bound is
   judgement SEQUENCE |- ID |-> EXP;

   prop_lookup_found: sequence[assign(I, val1).D] |- I |-> val1 ;

   prop_lookup_rec:
      D |- I |-> V
      ----------------
      sequence[assign(I', V').D] |- I |-> V ;
               provided id_diff(I, I');

   prop_lookup_not_found: sequence[] |- I |-> I ;
   end propag_bound;
end propag_eval;
```

D.3 Propagating constants through instructions: main specification

```
program propagation is
use little;
import glb(SEQUENCE + SEQUENCE =SEQUENCE + SEQUENCE + SEQUENCE) from glb;
import propag_eval(SEQUENCE |- EXP |-> EXP) from propag_eval;
import
   id_diff(ID, ID), val_diff(VAL, VAL), exp_diff(EXP, EXP),
   not_val(EXP), not_bool(EXP) from pl_misc;



export propagate_inst(sequence[] |- I -> I') as propagation(I) = I' ;
export propagate_program(|- P -> P') as propagate_program(P) = P' ;
var bool: BOOL;


set propagate_program is

judgement |- PROGRAM -> PROGRAM;

program_starter:
   propagation(sequence[] |- I -> I' + D')
   ----------------
   |- #program(D, I) -> #program(D, sequence[I'. D']) ;

end propagate_program;

set propagate_inst is

judgement SEQUENCE |- INST -> INST;

propag_starter:
   propagation(D |- I -> I' + D')
   ----------------
   D |- I -> sequence[I'. D'] ;

end propagate_inst;

set propagation is

judgement SEQUENCE |- INST -> INST + SEQUENCE;

propag_if_true:
   D |- I1 -> I' + D'
   ----------------
   D |- if(true(), I1, I2) -> I' + D' ;

propag_if_false:
   D |- I2 -> I' + D'
```

```
   ----------------
   D |- if(false(), I1, I2) -> I' + D' ;

propag_if_val:
   propag_eval(D |- E |-> bool)  &  D |- if(bool, I1, I2) -> I' + D'
   ----------------
   D |- if(E, I1, I2) -> I' + D' ;

propag_if_no_val:
   propag_eval(D |- E |-> E')  &  D |- I1 -> I1' + D1'  &  D |- I2 -> I2' + D2'  &
   glb(D1' + D2' =  D' + D1'' + D2'')
   ----------------
   D |- if(E, I1, I2) -> if(E', sequence[I1'.D1''],sequence[I2'.D2'']) + D' ;
            provided not_bool(E');

propag_while_false:
   propag_eval(D |- E |-> false())
   ----------------
   D |- while(E, I) -> sequence[] + D ; -- Here, there are many, many possible solutions,
                                       -- but an optimal one may be impossible to find.
                                       -- In this first version, we choose to do nothing.

propag_while:
   propag_eval(D |- E |-> V)
   ----------------
   D |- while(E, I) -> sequence[D, while(E, I)] + sequence[] ;
   provided exp_diff(V, false());

propag_sequence_end: D |- sequence[] -> sequence[] + D ;

propag_sequence_rec:
   D |- I1 -> I1' + D'  &  D' |- I2 -> I2' + D''
   ----------------
   D |- sequence[I1.I2] -> sequence[I1'.I2'] + D'' ;

propag_assign_val:
   propag_eval(D |- E |-> val)  &  partial_update(D |- I \ val -> D')
   ----------------
   D |- assign(I, E) -> sequence[] + D' ;

 propag_assign_not_val:
   propag_eval(D |- E |-> E')  &  remove(D |- I |-> D')
   ----------------
   D |- assign(I, E) -> assign(I, E') + D' ;
            provided not_val(E');

end propagation;

   set remove is
   judgement SEQUENCE |- ID |-> SEQUENCE;

   remove_end: sequence[] |- I |-> sequence[] ;
```

```
   remove_exact: sequence[assign(I, V).D] |- I |-> D ;

   remove_rec:
      D |- I |-> D'
      ----------------
      sequence[assign(I', V).D] |- I |->sequence[assign(I', V).D'] ;
               provided id_diff(I, I');
   end remove;

   set partial_update is
   judgement SEQUENCE |- ID \ VAL -> SEQUENCE;

   partial_update_exact:
      sequence[assign(I, _).D] |- I \ V ->sequence[assign(I, V).D] ;

   partial_update_rec:
      D |- I \ V -> D'
      ----------------
      sequence[assign(I', V').D] |- I \ V -> sequence[assign(I', V').D'] ;
               provided id_diff(I, I');

   partial_update_end: sequence[] |- I \ V -> sequence[assign(I, V)] ;
   end partial_update;


end propagation;
```

E Coq translation of the dynamic semantics

We give only as example the translation of the `dynamics` specification. The text is given in direct Coq syntax.

```
Require Export little. Require Export misc.
Require Export update. Require Export eval.

Mutual Inductive
   exec: DECLS -> INST -> DECLS ->Prop :=
        exec_assign:
          (E:EXP)
          (D', D:DECLS)
          (Id:ID) (V:VAL) (eval D E V) -> (update D Id V D') ->
          (exec D (assign Id E) D')
       | exec_if_true:
           (I1, I2:INST)
           (E:EXP)
           (D', D:DECLS)
           (eval D E (coer_BOOL_VAL l_true)) -> (exec D I1 D') ->
           (exec D (l_if E I1 I2) D')
       | exec_if_false:
           (I2, I1:INST)
           (E:EXP)
           (D', D:DECLS)
           (eval D E (coer_BOOL_VAL l_false)) -> (exec D I2 D') ->
           (exec D (l_if E I1 I2) D')
       | exec_while_false:
           (I:INST) (D:DECLS) (E:EXP) (eval D E (coer_BOOL_VAL l_false)) ->
           (exec D (while E I) D)
       | exec_while_true:
           (D', D'', D:DECLS)
           (E:EXP)
           (I:INST)
           (eval D E (coer_BOOL_VAL l_true)) ->
           (exec D I D') -> (exec D' (while E I) D'') ->
           (exec D (while E I) D'')
       | exec_sequence_end:
           (D:DECLS)(exec D (coer_SEQUENCE_INST (sequence (l_nil INST))) D)
       | exec_sequence_rec:
           (I2:(l_list INST))
           (D', D'', D:DECLS)
           (I1:INST)
           (exec D I1 D') ->
           (exec D' (coer_SEQUENCE_INST (sequence I2)) D'') ->
           (exec D (coer_SEQUENCE_INST (sequence (l_cons INST I1 I2))) D'').

Mutual Inductive
   dynamics: PROGRAM -> DECLS ->Prop :=
        start_program:
          (D', D:DECLS) (I:INST) (exec D I D') ->(dynamics (program D I) D').
```

F Proof of soundness of constant propagation

Lemma **update_diff:**
∀D : DECLS . ∀I : ID . ∀V : VAL . (bound D I V) ⇒
∀I' : ID . (id_diff I I') ⇒
∀D' : DECLS . ∀V' : VAL . (update D I' V' D') ⇒ (bound D' I V).

Lemma **update_diff_rev:**
∀D' : DECLS . ∀I : ID . ∀V : VAL . (bound D' I V) ⇒
∀I' : ID . (id_diff I I') ⇒
∀D : DECLS . ∀V' : VAL . (update D I' V' D') ⇒ (bound D I V).

Lemma **compatible_untouched:**
∀Id : ID . ∀S : SEQUENCE . (not_in Id S) ⇒
∀D, D' : DECLS .
∀Val : VAL .
(bound D Id Val) ⇒ (exec D (coer_SEQUENCE_INST S) D') ⇒
(bound D' Id Val).

Lemma **compatible_untouched_rev:**
∀Id : ID . ∀S : SEQUENCE . (not_in Id S) ⇒
∀D, D' : DECLS .
∀Val : VAL .
(exec D (coer_SEQUENCE_INST S) D') ⇒ (bound D' Id Val) ⇒
(bound D Id Val).

Inductive **wf_bindings : SEQUENCE ⇒ Prop** :=
wf_bindings_end: (wf_bindings (sequence (l_nil INST)))
| wf_bindings_rec:
∀Id : ID .
∀V : VAL .
∀Tl : (l_list INST) .
(wf_bindings (sequence Tl)) ⇒ (not_in Id (sequence Tl)) ⇒
(wf_bindings
(sequence (l_cons INST (assign Id (coer_VAL_EXP V)) Tl))).

Lemma **update_same:**
∀D, D' : DECLS . ∀I : ID . ∀V : VAL . (update D I V D') ⇒ (bound D' I V).

Lemma **propag_bound_val_or_eq:**
∀S : SEQUENCE .
∀Id : ID . ∀E : EXP . (propag_bound S Id E) ⇒ (wf_bindings S) ⇒
E = (coer_ID_EXP Id) ∨ ∃Val : VAL . E = (coer_VAL_EXP Val).

Lemma **propag_bound_correct:**
∀S : SEQUENCE .
∀Id : ID . ∀V : EXP . (propag_bound S Id V) ⇒ (wf_bindings S) ⇒
∀D' : DECLS . ∀V0 : VAL . (bound D' Id V0) ⇒
∀D0 : DECLS . (exec D0 (coer_SEQUENCE_INST S) D') ⇒ (eval D0 V V0).

Lemma **propag_eval_correct1:**
∀S : SEQUENCE . ∀E, V : EXP . (propag_eval S E V) ⇒ (wf_bindings S) ⇒
∀D' : DECLS . ∀V0 : VAL . (eval D' E V0) ⇒
∀D0 : DECLS . (exec D0 (coer_SEQUENCE_INST S) D') ⇒ (eval D0 V V0).

Lemma **update_commute:**
∀d, d' : DECLS . ∀id : ID . ∀V : VAL . (update d id V d') ⇒
∀id' : ID . (id_diff id id') ⇒
∀d'' : DECLS . ∀V' : VAL . (update d id' V' d'') ⇒
∃d3 : DECLS . (update d' id' V' d3) ∧ (update d'' id V d3).

Lemma **update_commute2:**
∀Id : ID . ∀D, D' : DECLS . ∀V : VAL . (update D Id V D') ⇒
∀Id' : ID . (id_diff Id Id') ⇒
∀D'' : DECLS . ∀V' : VAL . (update D' Id' V' D'') ⇒
∃D2 : DECLS . (update D Id' V' D2) ∧ (update D2 Id V D'').

Lemma **update_not_in:**
∀S : SEQUENCE . (wf_bindings S) ⇒
∀Id : ID . (not_in Id S) ⇒
∀D', D'' : DECLS . (exec D' (coer_SEQUENCE_INST S) D'') ⇒
∀D : DECLS . ∀V : VAL . (exec D (assign Id (coer_VAL_EXP V)) D') ⇒
∃D2 : DECLS .
(exec D (coer_SEQUENCE_INST S) D2) ∧
(exec D2 (assign Id (coer_VAL_EXP V)) D'').

Lemma **update_not_in_rev:**
∀S : SEQUENCE . (wf_bindings S) ⇒
∀Id : ID . (not_in Id S) ⇒
∀D', D'' : DECLS .
∀V : VAL . (exec D' (assign Id (coer_VAL_EXP V)) D'') ⇒
∀D : DECLS . (exec D (coer_SEQUENCE_INST S) D') ⇒
∃D2 : DECLS .
(exec D (assign Id (coer_VAL_EXP V)) D2) ∧
(exec D2 (coer_SEQUENCE_INST S) D'').

Lemma **glb1notin:**
∀B1, B2, B', B1', B2' : SEQUENCE .
∀Id : ID . (glb B1 B2 B' B1' B2') ⇒ (not_in Id B1) ⇒ (not_in Id B').

Lemma **get_binding_not_not_in:**
∀d, d' : SEQUENCE .
∀id : ID .
∀v : EXP . ∀Inst : INST . (get_binding d Inst d') ⇒ Inst = (assign id v) ⇒
¬ (not_in id d).

Lemma **get_binding_not_in_trans:**
∀d, d' : SEQUENCE .
∀Inst : INST . ∀Id : ID . (get_binding d Inst d') ⇒ (not_in Id d) ⇒
(not_in Id d').

Lemma **glb2notin:**
∀B1, B2, B', B1', B2' : SEQUENCE .
∀Id : ID . (glb B1 B2 B' B1' B2') ⇒ (not_in Id B2) ⇒ (not_in Id B').

Lemma **glb_not_in_transmit1:**
∀D1 : SEQUENCE . ∀I : ID . (not_in I D1) ⇒
∀D2, R, R1, R2 : SEQUENCE . (glb D1 D2 R R1 R2) ⇒ (not_in I R1).

Lemma **glb_not_in_transmit2:**
∀D1, D2, R, R1, R2 : SEQUENCE . (glb D1 D2 R R1 R2) ⇒
∀I : ID . (not_in I D2) ⇒ (not_in I R2).

Lemma **glb_wf1:**
∀D1 : SEQUENCE . (wf_bindings D1) ⇒
∀D2, R, R1, R2 : SEQUENCE . (glb D1 D2 R R1 R2) ⇒ (wf_bindings R1).

Lemma **get_binding_wf:**
∀s, s' : SEQUENCE . ∀i : INST . (get_binding s i s') ⇒ (wf_bindings s) ⇒
(wf_bindings s').

Lemma **get_binding_not_in_both:**
∀s : SEQUENCE . (wf_bindings s) ⇒
∀s' : SEQUENCE . ∀id : ID . ∀v : EXP . (get_binding s (assign id v) s') ⇒
(not_in id s').

Lemma **glb_wf2:**
∀D1, D2, R, R1, R2 : SEQUENCE .
(glb D1 D2 R R1 R2) ⇒ (wf_bindings D2) ⇒ (wf_bindings R2).

Lemma **get_binding_decompose:**
∀S, S' : SEQUENCE .
∀I : ID . ∀V : EXP . (get_binding S (assign I V) S') ⇒ (wf_bindings S) ⇒
∀D, D' : DECLS . (exec D (coer_SEQUENCE_INST S) D') ⇒
∃D2 : DECLS .
(exec D (coer_SEQUENCE_INST S') D2) ∧ (exec D2 (assign I V) D').

Lemma **get_binding_val:**
∀s : SEQUENCE . (wf_bindings s) ⇒
∀s' : SEQUENCE . ∀id : ID . ∀v : EXP . (get_binding s (assign id v) s') ⇒
∃V : VAL . v = (coer_VAL_EXP V).

Lemma **glb_wf:**
∀D1 : SEQUENCE . (wf_bindings D1) ⇒
∀D2, R, R1, R2 : SEQUENCE . (glb D1 D2 R R1 R2) ⇒ (wf_bindings R).

Lemma **glb2_wf:**
∀D1, D2, R, R1, R2 : SEQUENCE .
(glb D1 D2 R R1 R2) ⇒ (wf_bindings D2) ⇒ (wf_bindings R).

Lemma **glb1_correct:**
∀S1, S2, S, S'1, S'2 : SEQUENCE .
(glb S1 S2 S S'1 S'2) ⇒ (wf_bindings S1) ⇒
∀D, D1 : DECLS . (exec D (coer_SEQUENCE_INST S1) D1) ⇒
∃D2 : DECLS .
(exec D (coer_SEQUENCE_INST S'1) D2) ∧
(exec D2 (coer_SEQUENCE_INST S) D1).

Lemma **glb2correct:**
∀S1, S2, S, S'1, S'2 : SEQUENCE .
(glb S1 S2 S S'1 S'2) ⇒ (wf_bindings S2) ⇒
∀D, D1 : DECLS . (exec D (coer_SEQUENCE_INST S2) D1) ⇒
∃D2 : DECLS .
(exec D (coer_SEQUENCE_INST S'2) D2) ∧
(exec D2 (coer_SEQUENCE_INST S) D1).

Lemma **remove_not_in:**
∀Id : ID . ∀S, S' : SEQUENCE . (remove S Id S') ⇒ (wf_bindings S) ⇒
(not_in Id S').

Lemma **not_in_remove:**
∀Id : ID . ∀S : SEQUENCE . (not_in Id S) ⇒
∀S' : SEQUENCE . ∀Id' : ID . (remove S Id' S') ⇒ (not_in Id S').

Lemma **remove_wf:**
∀S : SEQUENCE . (wf_bindings S) ⇒
∀Id : ID . ∀S' : SEQUENCE . (remove S Id S') ⇒ (wf_bindings S').

Lemma **propag_bound_remove_diff:**
∀S : SEQUENCE . ∀Id : ID . ∀E : EXP . (propag_bound S Id E) ⇒
∀Id' : ID . (id_diff Id Id') ⇒
∀S' : SEQUENCE . (remove S Id' S') ⇒ (propag_bound S' Id E).

Lemma **not_in_propag_bound:**
∀S : SEQUENCE . ∀Id : ID . (not_in Id S) ⇒
(propag_bound S Id (coer_ID_EXP Id)).

Lemma **propag_bound_remove_eq:**
∀S, S' : SEQUENCE . ∀Id : ID . (remove S Id S') ⇒ (wf_bindings S) ⇒
(propag_bound S' Id (coer_ID_EXP Id)).

Lemma **update_twice:**
∀d, d' : DECLS . ∀v : VAL . ∀id : ID . (update d id v d') ⇒
∀d'' : DECLS . ∀v' : VAL . (update d' id v' d'') ⇒ (update d id v' d'').

Lemma **update_trans:**
∀d, d' : DECLS . ∀id : ID . ∀v : VAL . (update d id v d') ⇒
∀id' : ID . ∀v' : VAL . ∀d0 : DECLS . (update d0 id' v' d) ⇒
∃d'' : DECLS . (update d0 id v d'').

Lemma **exec_assign_val:**
∀d, d' : DECLS . ∀id : ID . ∀val : VAL . (update d id val d') ⇒
(exec d (assign id (coer_VAL_EXP val)) d').

Lemma **remove_not_in_trans:**
∀S : SEQUENCE . (wf_bindings S) ⇒
∀Id : ID . (not_in Id S) ⇒
∀Id' : ID . ∀S' : SEQUENCE . (remove S Id' S') ⇒ (not_in Id S').

Lemma **remove_correct:**
∀S, S' : SEQUENCE . ∀Id : ID . (remove S Id S') ⇒ (wf_bindings S) ⇒
∀D, D' : DECLS . (exec D (coer_SEQUENCE_INST S) D') ⇒
∀D'' : DECLS . ∀V : VAL . (update D' Id V D'') ⇒
∃D2 : DECLS .
(update D Id V D2) ∧ (exec D2 (coer_SEQUENCE_INST S') D'').

Lemma **partial_update_not_in:**
∀Id : ID . ∀V : VAL . ∀S, S' : SEQUENCE . (partial_update S Id V S') ⇒
∀Id' : ID . (id_diff Id' Id) ⇒ (not_in Id' S) ⇒ (not_in Id' S').

Lemma **partial_update_wf:**
∀Id : ID .
∀V : VAL .
∀S, S' : SEQUENCE . (partial_update S Id V S') ⇒ (wf_bindings S) ⇒
(wf_bindings S').

Lemma **propagation_wf:**
∀I, I' : INST .
∀S, S' : SEQUENCE . (propagation S I I' S') ⇒ (wf_bindings S) ⇒
(wf_bindings S').

Lemma **partial_update_correct1:**
∀s, s' : SEQUENCE .
∀id : ID . ∀val : VAL . (partial_update s id val s') ⇒ (wf_bindings s) ⇒
∀d, d1, d2 : DECLS .
(exec d (coer_SEQUENCE_INST s) d1) ⇒ (update d1 id val d2) ⇒
(exec d (coer_SEQUENCE_INST s') d2).

Lemma **propagation_correct1:**
∀S, S' : SEQUENCE .
∀I, I' : INST . (propagation S I I' S') ⇒ (wf_bindings S) ⇒
∀D, D1 : DECLS . (exec D (coer_SEQUENCE_INST S) D1) ⇒
∀D' : DECLS . (exec D1 I D') ⇒
∃D2 : DECLS . (exec D I' D2) ∧ (exec D2 (coer_SEQUENCE_INST S') D').

Lemma **bound_not_in:**
∀I : ID . ∀S : SEQUENCE . (not_in I S) ⇒
∀D : DECLS . ∀V : VAL . (bound D I V) ⇒
∀D' : DECLS . (exec D (coer_SEQUENCE_INST S) D') ⇒ (bound D' I V).

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803